Vx960™

PROGRAMMER'S
GUIDE

Intel Corporation

Contents

Documentation Guide

Vx960 documentation is divided into the following manuals:

- The *Vx960 Programmer's Guide* describes the components of the Vx960 system and how they interrelate.
- The *Vx960 Reference Manual* contains UNIX-style manual page entries for all Vx960 modules and subroutines.
- The *Board Support Package Manual* describes each board in man page format.
- The *GNU/960 Manual* contains the following documentation:
  - The *GNU/960 Tools Release Notice* provides an overview of the highlights of the GNU/960 release. It covers the differences between the GNU/960 release and previous releases, and it includes information on fixes to known bugs.
  - *Installing the GNU/960 Tools* gives detailed step-by-step installation information for the GNU/960 tools (source, binaries, and documentation).
  - *Using the GNU/960 Tools* is an entry-point document that describes the history and usage of the tools, gives a summary overview of each command in the tool set, and explains the concepts and process of developing an application using the GNU/960 tools.
  - The *GNU/960 Tool Development* manual describes how to customize the GNU/960 tools for a variety of purposes, including porting to other hosts.
  - The *GNU/960 Reference Manual* contains the man pages which provide detailed information on each GNU/960 command. Many man pages also provide feature examples. The **intro(1GNU)** man page gives an overview of all the GNU/960 tools.

- *Using Intel i960 GNU CC* provides in-depth information on gcc960, the GNU/960 C compiler, including detailed descriptions of all commandline options. This is the source for complete reference information on gcc960.
- *Using GDB/960* provides in-depth information on gdb960, the GNU/960 debugger, including detailed descriptions of all commandline options. This is the source for complete reference information on gdb960. A gdb960 quick reference is also included.

Note: For additional information on the gdb-based debugger available for Vx960, refer to the *Vx960 Release Notes*.

- *Using GAS/960* provides in-depth information on gas960, the GNU/960 assembler, including detailed descriptions of all commandline options. This is the source for complete reference information on gas960.
- *Using GLD/960* provides in-depth information on gld960, the GNU/960 linker including detailed descriptions of all commandline options. This is the source for complete reference information on gld960.

Vx960 Programmer's Guide

The *Vx960 Programmer's Guide* contains the following chapters:

- **Chapter 1. Overview**
  Outlines all Vx960 facilities and subsystems, and indicates where to find additional information about each.

- **Chapter 2. Getting Started**
  Provides step-by-step procedures for:
  - installing Vx960
  - configuring host software and target hardware and software
  - booting Vx960
  - loading and running the demo programs.

- **Chapter 3. Basic OS**
  Describes the fundamentals of the Vx960 run-time environment including:
  - multitasking kernel
  - spawning and manipulation of tasks
  - intertask communication facilities
  - interrupt service code.

- **Chapter 4. I/O System**
  Describes the user view of the Vx960 I/O system in general, the specifics of Vx960 supplied device drivers, and the internal details of the I/O system, including how to write device drivers.

- **Chapter 5. Local File Systems**
  Describes available local file systems: a DOS-compatible file system, an RT-11 file system, and a raw file system.

- **Chapter 6. Network**
  Describes the Vx960 network and remote file system. Covers TCP/IP, Ethernet, backplane network, remote procedure calls, and network file systems.

- **Chapter 7. Cross-Development**
  Discusses Vx960 cross-development facilities and procedures. Describes the module loader and system symbol table, how to build and load application modules, and how to configure and build Vx960 itself.

- **Chapter 8. Configuration**
  Discusses the Vx960 standard configuration, plus the various options that can be exercised to tailor the system to particular hardware and software requirements.

- **Chapter 9. Shell**
  Describes the Vx960 shell, an interactive C-expression interpreter.

- **Chapter 10. Debugging**
  Describes Vx960 debugging facilities and techniques.

- **Appendix A. Directories and Files**
  Describes Vx960 directories and files, showing the Vx960 system directory structure and summarizing the role of each component.

- **Appendix B. Memory Layout**
  Provides a map of system memory layout.

- **Appendix C. Coding Conventions**
  Describes coding conventions used by Intel in all source modules.

- **Appendix D. Bibliography**
  Provides a collection of resources on internetworking, operating systems in general, and UNIX in particular. This bibliography is meant as a starting point for further research.

- **Index**
  Provides an extensive topical index.

Vx960 Reference Manual

The *Vx960 Reference Manual* consists of UNIX-style man page entries, and it is divided into the sections summarized below.

- **Master Intex**
  Contains the table of contents for Vx960 libraries and routines.

- **Libraries (1)**
  Contains manual entries for all Vx960 libraries. Each manual entry lists the routines found in a given library, including a one-line synopsis of each, along with a general description of their use. Individual manual entries for each subroutine follow the general description. Entries for libraries that are target-specific (e.g., sysLib and sysAlib) are found in the Supplements section.

- **Drivers (2)**
  Contains manual entries for all Vx960 supplied drivers. Entries for drivers that are target-specific (e.g., tyCoDrv) are found in the Supplements section.

- **Tools (3)**
  Contains manual entries for all Vx960 tools that run under UNIX.

Target Supplements

The following supplements are supplied separately. These supplements provide information specific to each target supported by Vx960 and contain the manual entries for the target-specific modules. The following is a list of currently supported targets for which the supplements are available:

Cyclone CVME960	Intel EV80960CA
Heurikon HK80/V960E	Intel EV80960SX
Tadpole TP-960V	Intel TomCAT

Note: The Intel TomCAT board is not commercially available at this time. It is included in the list above for completeness.

The targets in the first column are commercial Versa Module Eurocards (VMEs). The Intel targets listed in the second column are Intel processor evaluation boards. For additional information on each of these, refer to the *Solutions 960* catalogue.

To obtain the *Solutions 960* catalogue write to:

Intel Literature Sales
P.O. Box 7641
Mt. Prospect, IL 60056-7641

For phone orders in the U.S. and Canada call toll free: (800) 548-4725.

On-Line Documentation

The Vx960 distribution includes the **nroff** sources for the entire *Vx960 Reference Manual*. Formatted entries can be accessed on-line from the host using the tool **vxman**(3). For more information, see **2.7.2 On-Line Help** and the manual entry for **vxman**(3).

Typographical Conventions

Vx960 printed documentation uses the font conventions shown in the table below to differentiate various elements. In addition, parentheses are always included with subroutine names, and a caret (^) is used to indicate control characters. Also, when libraries, drivers, or tools are cross-referenced to a section of the *Vx960 Reference Manual*, the section number is shown in parentheses after the name, e.g., taskLib(1).

Font Style for Special Terms	
Term	**Example**
files, pathnames	/etc/hosts
libraries, drivers	**memLib.c, nfsDrv.c**
UNIX shell tools	**vxman**
subroutines	*semTake( )*
boot commands	**p**
code display	`main ();`
keyboard input	`rlogin vx`
display output	`value = 0`
user-supplied parameters	*name*
constants	INCLUDE_NFS
C directives/data types	*#define*
keyboard return	RETURN
control characters	^C
lower-case acronyms	*fd*

1

Overview

Contents

1

Overview

1.1 Introduction

Vx960 is a high-performance, real-time operating system and a powerful development environment for real-time applications. Vx960 includes a fast and flexible run-time system, powerful testing and debugging facilities, and an unparalleled UNIX cross-development package, at the heart of which lies Vx960's extensive UNIX-compatible networking facilities.

The networking facilities allow Vx960 and UNIX to combine to form a complete, integrated development and operational environment. Each is used for what it does best. The UNIX system is used for software development and the non-real-time components of applications, while Vx960 is used for testing, debugging, and running real-time applications.

Once development is complete, the Vx960 system can operate either stand-alone or networked with other systems running Vx960 or UNIX.

Vx960 is available for several target intel960™ microprocessor, CPU boards and can be networked with any BSD 4.2 or 4.3 UNIX host systems, or any other operating system with TCP/IP networking facilities.

The Vx960 real-time system includes:

High-Performance Real-Time Kernel Facilities
Multitasking with preemptive priority scheduling, intertask synchronization and communications facilities, interrupt handling support, watchdog timers, and memory management.

Network Facilities Transparent access to other Vx960 and UNIX systems via UNIX source-compatible sockets, remote command execution, remote login, remote procedure calls (RPC), source-level remote debugging, and remote file access, all using TCP/IP network protocols both loosely-coupled over standard Ethernet connections and tightly-coupled over a backplane bus using shared memory.

Module Loader and System Symbol Table
Dynamic loading of object modules over the network or from a disk, with run-time relocation and linking.

Shell A C-interpreter interface that allows interactive execution of most C language expressions, Vx960 functions, and any other loaded functions, and also includes symbolic references to variables.

Debugging Facilities
Source-level debugging, a symbolic disassembler, symbolic C-subroutine traceback, task-specific breakpoints and single-stepping, system status displays, and exception handling to fault and report on interrupts and hardware exceptions such as bus or address errors.

I/O System A fast and flexible UNIX-source-compatible I/O system, including UNIX standard buffered I/O.

Local File Systems
Fast file systems appropriate for real-time and compatible with the MS-DOS and RT-11 file systems, as well as a raw disk file system.

Remote File System
Network File System (NFS) facilities for accessing files transparently on any NFS server on the network, and a non-NFS network facility for accessing the host file systems using *rsh* or *ftp*.

Performance Evaluation Tools
An execution timer for timing a routine or group of routines, and utilities to show CPU utilization percentage by task.

Utility Libraries
An extensive set of utility functions available to application developers, including: message logging, string formatting and scanning, linear and ring buffer manipulations, linked-list manipulations, and symbol table manipulation.

I/O Drivers

Ty driver	for serial I/O devices
Network driver	for remote files
Pipe driver	for intertask communication
RAM "disk" driver	for memory resident files
SCSI library	for SCSI hard disks and floppies.

Board-Support Packages
Routines for hardware initialization, interrupt setup, timers, memory mapping, etc.

Boot ROMs
Allow a target CPU to be booted over the network.

System Configuration Utilities
Allow reconfiguration and extension of Vx960 and building applications in ROM.

1.2 Vx960 A Real-Time Partner for UNIX

The UNIX Operating System is an excellent system for program development and for many interactive applications. However, it does not support real-time applications well. On the other hand, traditional real-time operating systems provide poor environments for program development or for non-real-time components of an application.

Rather than create a single operating system that does it all, Intel has networked two different, but cooperating operating systems, Vx960 and UNIX. Vx960 handles the critical real-time chores, while UNIX is used for program development and for non-time-critical applications. Vx960 and UNIX work well together because Vx960 has been designed to be UNIX-compatible at many levels, in its extensive networking facilities.

As a cross-development host, UNIX is used to edit, compile, link, and store real-time code, which is run and debugged on Vx960. The resulting Vx960 application can be run stand-alone either in ROM or disk based, with no further need for the network or the host system.

However, UNIX and Vx960 can also work together in a hybrid application, with UNIX systems using Vx960 systems as real-time servers in a networked environment. For instance, a Vx960 system controlling a robot might itself be controlled by the UNIX system running an expert system, or a number of Vx960 systems running factory equipment might be connected to UNIX systems tracking inventory or generating reports.

1.3 Development Cycle

To help you understand the environment provided by Vx960, we have outlined a typical development cycle. The hardware in a typical development environment includes one or more multi-user UNIX host systems and one or more single-user Vx960 target systems connected by an Ethernet network. The UNIX system can be loaded with large main memory, large disks, backup media, printers, and terminals.

The target systems, on the other hand, have only the resources required by the real-time system, plus some for testing and debugging. This can be as little as a CPU, some serial I/O channels, and an Ethernet connection.

Software development for a real-time system begins on the UNIX host development system. Using the development and management tools on UNIX, the application team begins to design and implement the application modules. Developers are free to use the usual UNIX tools such as text editors, compilers, assemblers, **make**, source-code control, and so on. The applications themselves can make use of the many libraries supplied by Vx960.

Application modules in C are compiled in the usual way with a cross-compiler. The application modules do not need to be linked with the Vx960 system libraries or even with each other. Instead, Vx960 can load the UNIX-generated object modules, using the symbol table contained in all object modules to resolve external symbol references.

Selected modules can be dynamically loaded across the network for testing and debugging. The Vx960 "shell" program can then be used to invoke and test individual application subroutines, or complete tasks.

Vx960 remembers the symbol tables from previously-loaded object modules, giving symbolic access to data and subroutine names. You can examine data variables, call subroutines, spawn tasks, disassemble code in memory, set breakpoints, obtain subroutine call tracebacks, and so on, all using the original symbolic names. Also, program errors detected by the hardware, such as illegal memory references or illegal instructions, are faulted and reported by Vx960, allowing further symbolic debugging.

Source-level debuggers are available that allow the application to be viewed and debugged in the original source code. Setting breakpoints, single-stepping, examining variables, and so on, can be done at the source level, using either commands at an ASCII terminal or a mouse-based menu-driven interface on a windowed workstation.

The cycle of building, downloading, and testing modules is iterated until the application is ready for the production environment. You can remove Vx960 debugging facilities from the production system, if necessary, to produce a system requiring minimal resources. At that point, you can link the application with Vx960, and put it into ROM if desired.

The remainder of this chapter outlines each of the components listed in the overview above and provides reference to further documentation.

1.4 Multitasking and Intertask Communications

Modern real-time systems are based on the complementary concepts of multitasking and intertask communications. A multitasking environment allows real-time applications to be constructed as set of independent tasks, each with its own thread of execution and set of system resources. The intertask communication facilities allow these tasks to synchronize and communicate in order to coordinate their activity.

The Vx960 multitasking kernel uses interrupt-driven, priority-based task scheduling. It features fast, context-switch times and low, interrupt latency.

Under Vx960, any C subroutine can be "spawned" as a separate task, with its own context and stack. Other basic task-control allows tasks to be suspended, resumed, deleted, delayed, and moved in priority. See **3.2 Tasks** in the **Basic OS** chapter as well as the manual entry for taskLib(1).

Vx960 supplies several types of traditional task-blocking semaphores as the basic task synchronization and mutual-exclusion mechanism. Vx960 semaphores are fast and efficient. In addition to being available to application builders, they have been used in building higher level facilities in Vx960.

For intertask communications, Vx960 supplies a fast and flexible message queue facility, intertask pipes, sockets, and signals. Sockets are a UNIX-compatible mechanism for exchanging byte streams between tasks regardless of location in a networked application. Signals are a UNIX-compatible mechanism for asynchronous transfer of control within a task based on hardware or software exceptions. For additional information see the following:

- semaphores are described in the manual entry for semLib(1) and in Chapter 3. Basic OS;
- message queues are described in msgQLib(1) and in Chapter 3. Basic OS;
- pipes are described in pipeDrv(2) and in Chapters 3. Basic OS and 4. I/O System;
- sockets are described in sockLib(1) and in Chapters 3. Basic OS and 6. Network;
- signals are described in sigLib(1) and in Chapter 3. Basic OS.

1.5 Network

The key to Vx960's partnership with UNIX is its extensive networking facilities. Since the network provides a fast, easy-to-use connection between the two systems, you can use UNIX as a development system, as a debugging host, and as a provider of non-real-time services in a final system.

Vx960 supports network connections both loosely-coupled over Ethernet networks (IEEE 802.3), and tightly-coupled over a backplane bus using shared memory. Vx960 uses the TCP/IP network protocols as implemented in BSD 4.3 for all network communications.

Vx960 provides several levels of network access: process-to-process sockets, remote command execution, remote login, remote procedure calls, remote file access, and remote source-level debugging.

1.5.1 Sockets

Vx960 provides standard UNIX socket calls, which allow real-time Vx960 processes and other processes, such as UNIX processes, to communicate in any combination with each other over the network. Vx960 socket calls are source compatible with UNIX BSD 4.3.

Any process can open one or more sockets to which other sockets can be connected. Data written to one socket of a connected pair can be read from the other socket. The network link is transparent in the communications. In fact, the two processes do not know whether they are communicating with another process on the same CPU or another CPU, or with a Vx960 process or a UNIX process.

See the manual entry for **sockLib**(1).

1.5.2 Remote File Access: NFS, ftp, rsh

Remote file access across the network is also available. A program running on Vx960 is able to use a UNIX system as a "virtual file system." Files on any UNIX system can be accessed, via the network, as if they were local to the Vx960 system. A program running under Vx960 does not need to know where that file is or how to access it. For example, **/dk/file** might be a file local to the Vx960 system, while **host:file** might be a file located on another machine.

Vx960 includes the SUN Microsystems standard Network File System (NFS). It runs as an NFS client with any other system that runs an NFS server. Vx960 can also use two older protocols to provide transparent remote file access: *rsh*, or *ftp*. An *ftp* server provides remote access to Vx960 from other Vx960 or UNIX systems using *ftp*.

See the manual entries for **nfsLib**(1), **remLib**(1), **ftpLib**(1), and **ftpdLib**(1), and the sections **4.6.4 Network File System (NFS) Devices** and **4.6.5 Non-NFS Network Devices** in the **I/O System** chapter.

1.5.3 Remote Login: rlogin, telnet

The remote login feature allows you to log into Vx960 or UNIX machines from any other Vx960 or UNIX machine on the network. On a UNIX workstation, you can open an **rlogin** window that communicates with the Vx960 shell. By opening such windows, you can monitor and control real-time Vx960 systems right from their desks.

Vx960 can also be accessed via **telnet**, for systems that do not have **rlogin**.

See the manual entries for **rlogLib**(1) and **telnetLib**(1).

1.5.4 Remote Procedure Calls (RPC)

Designed by SUN Microsystems and available in the public domain, Remote Procedure Call (RPC) is a facility that allows a process on one machine to call a procedure which is executed by another process on another machine. Thus with RPC, a Vx960 task or UNIX process can invoke routines that are executed on other Vx960 or UNIX machines, in any combination. See the public domain RPC documentation and the manual entry for **rpcLib**(1).

1.6 Module Loader and System Symbol Table

The communication mechanism from application to Vx960 real-time, operating system is a subroutine call. Vx960 does not need mechanisms such as system faults to get to system functions. Instead Vx960 supplies a system symbol table and a loader with run-time linking to give dynamic and even interactive access to all loaded modules.

The Vx960 module loader can load object modules over the network or from a disk, and relocate them anywhere in memory. The loader also uses the symbol table contained in every object module to build a system-wide symbol table of loaded function and variable names. Names from system and application modules alike are added to the system symbol table.

This symbol table is the heart of many of Vx960's most significant development aids. The loader itself uses the system symbol table to resolve undefined references in modules being loaded, linking newly loaded modules to previously loaded modules. Also, Vx960 uses the system symbol table to provide interactive access to all system and application modules that have been loaded. Finally, all of Vx960's debugging facilities use the system symbol table to provide symbolic references wherever possible.

Run-time linking makes it easy to have shared subroutine libraries, in which a single copy of a set of subroutines can be used by several tasks, rather than requiring each task to be linked with separate copies of needed subroutines. As a result, there is no inherent distinction between Vx960 "system" modules and user "application" modules. This makes the Vx960 real-time system an open system: the system facilities are easy to access, modify, and extend.

For more information on the module loader and the system symbol table, see **7. Cross-Development**. Also see the manual entries for loadLib(1) and symLib(1).

1.7 Shell

Vx960 includes an interactive program called the "shell," which allows developers to interact with all Vx960 facilities. The Vx960 shell provides one simple but powerful capability: it can interpret and execute almost all C-language expressions, including calls to functions and references to variables whose names are found in the system symbol table.

Thus the shell can be used to call Vx960 system functions, call any application functions, examine and set application variables, create new variables, and even as a general purpose calculator with all C operators.

In addition, the shell includes a command history facility and **vi**-like command-line editing, and can also be used to log into a remote UNIX or Vx960 machine with **rlogin**.

The Vx960 shell is discussed in detail in **9. Shell**. See also the manual entries for **shellLib**(1), **usrLib**(1), and **dbgLib**(1), which contain routines suited for interactive access of Vx960 facilities.

1.8 Debugging Facilities

Vx960 supplies a powerful set of debugging facilities. These include:

- routines to display system and task status
- a symbolic disassembler that can disassemble any loaded module
- a symbolic C-subroutine traceback facility that can be called at any time to list the current sequence of nested subroutine calls of any task
- complete trapping of hardware exceptions in a non-fatal way that allows symbolic debugging to continue
- a breakpoint and single-stepping facility that can be applied to specific tasks, even in shared code.

As noted above, all these facilities use the system symbol table to provide symbolic references wherever possible.

For more information on Vx960's debugging facilities, see **10. Debugging**. See also the manual entries for **usrLib**(1), **dbgLib**(1), **excLib**(1), and **dsmLib**(1).

In addition, the Vx960 distribution includes a port for a gdb-based, source-level debugger, an extended version of the GNU Source-Level Debugger (GDB). This allows full source-level debugging of remote Vx960 applications from a variety of host UNIX systems.

Using the gdb-based debugger, you can spawn and debug tasks that are running on networked Vx960 targets. You can also debug already-running tasks spawned from the Vx960 shell. While using this debugger, you can continue to take advantage of Vx960's native development tools. By combining the Vx960 shell, symbolic debugging and disassembly, and performance monitoring facilities with the gdb-based debugger capabilities, you have a comprehensive high-level debugging solution.

For additional information on the gdb-based debugger, refer to the *Vx960 Release Notes*.

1.9 Performance Evaluation

To understand and optimize the performance of a real-time system, you must time various functions that the system performs. Vx960 provides various timing facilities to help with this task.

The Vx960 execution timer is able to time any C subroutine or group of subroutines. Because the system clock is too slow to provide the resolution necessary to time fast functions, the timer is also able to iterate execution of a group of functions until the time of a single iteration is known to a reasonable tolerance. For more information on the execution timer, see the manual entry for **timexLib**(1).

Vx960 provides a utility that shows, for each task, the amount of CPU time utilized, the amount of time spent at interrupt level, and the amount of idle time. Time is displayed in ticks and in percentage. For more information, see the manual entry for **spyLib**(1).

1.10 I/O System

The Vx960 I/O system provides uniform device-independent access to many kinds of devices. The user can call seven basic I/O functions: *creat( )*, *delete( )*, *open( )*, *close( )*, *read( )*, *write( )*, and *ioctl( )*. Higher-level I/O functions, such as UNIX-compatible *printf( )* and *scanf( )* routines, are provided and built on these basic functions.

Vx960 also provides an *stdio* buffered I/O package that includes UNIX-compatible routines such as *fopen( )*, *fclose( )*, *fread( )*, *fwrite( )*, *getch( )*, *putch( )*, etc. These routines increase I/O performance in many cases.

Vx960 includes device drivers for serial communications lines, disks, "RAM disks," intertask communication devices called pipes, and devices on a network. Application developers can write additional drivers. Vx960 allows dynamic installation and removal of drivers without rebooting the system.

The Vx960 I/O system is fast and flexible, allowing individual drivers complete control over how the user requests are serviced. Drivers can implement different protocols, unique device-specific functions, and even different file systems, without interference from the I/O system itself. Vx960 also supplies several high-level packages that make it easy for drivers to implement common device protocols and file systems.

For a detailed discussion of the I/O system, see **4. I/O System**. Relevant manual entries include **ioLib**(1) for basic I/O routines available to user tasks, **fioLib**(1) and **stdioLib**(1) for various format-driven I/O routines, and **iosLib**(1) and **tyLib**(1) for routines available to driver writers. Also see the manual entries for the supplied drivers.

1.11 Local File Systems

Vx960 includes different local files systems for use with block devices (disks). These devices all use a standard interface so that file systems can be mixed with device drivers. Vx960 I/O architecture makes it possible to have several different file systems, even at the same time, on a single Vx960 system.

1.11.1 DOS File System

Vx960 offers a file system compatible with DOS for personal computers. Vx960 DOS is compatible with versions of MS-DOS up to and including 4.0. Vx960 DOS capabilities offer flexibility appropriate to the varying demands of real-time applications. Major features include:

- A hierarchical arrangement of files and directories, allowing efficient organization and permitting an indefinite number of files to be created on a volume.
- A choice of file fragmentation or contiguity on a per-file basis. File fragmentation results in more efficient use of available disk space while contiguity offers enhanced performance.
- Compatibility with available storage and retrieval media. Disks created with Vx960 DOS and on DOS personal computers can be interchanged.

The Vx960 DOS file system is implemented in dosFsLib(1).

1.11.2 RT-11 File System

Vx960 is supplied with a file system compatible with that of the RT-11 operating system. This file system is appropriate for many real-time file systems, since all files are contiguous. File accesses require one disk access, and sequential file accesses involve minimal disk movement.

The RT-11 file system lacks a hierarchical file organization that is useful on large disks. Also, the contiguous allocation scheme can result in fragmented disk space.

The Vx960 implementation of the RT-11 file system includes byte-addressable random access (seeking) to all files. Each open file has a block buffer for optimized reading and writing. The RT-11 file system is implemented in **rt11FsLib**(1).

1.11.3 Raw Disk File System

Vx960 offers a simple "raw file system" for use with disk devices. The raw file system treats the entire disk much like a single large file. Portions of the disk can be read and written, specified by byte offset, and simple buffering is performed. The raw file system offers advantages of size and speed when low-level disk I/O is required.

This functionality was available by using the RT-11 file system and specifying the device name as a file name for file operations. Services for file-oriented device drivers using the raw file system are implemented in rawFsLib(1).

1.11.4 Alternative File Systems

In Vx960, the file system is not tied to the device or its driver. A device can be associated with any file system. Alternative, user-supplied file systems can be written and used by drivers in the same way, by following the same standard interfaces between the file system, the driver, and the Vx960 I/O system.

1.12 Utility Libraries

Vx960 supplies many subroutines of general utility to application developers. These routines are organized as a set of subroutine libraries, which are described below. Application developers are encouraged to use these libraries wherever possible. Doing so reduces both development time and memory requirements for the application.

Interrupt Handling Support

Vx960 supplies routines for handling hardware interrupts and faults without having to resort to assembly language coding. Routines are provided to connect C routines to hardware interrupt vectors, and to manipulate the processor interrupt level.

See the manual entries for **intLib**(1) and **intALib**(1) for more information. Also see **3. Basic OS** for information about the context in which interrupt level code runs, and for special restrictions that apply.

Watchdog Timers

A watchdog facility allows callers to schedule execution of their own routines after specified time delays. As soon as the specified number of ticks has elapsed, the specified timeout routine is called at the interrupt level of the system clock, unless the watchdog is canceled first. Note that this mechanism is different from the kernel's task delay facility.

See the chapter **3. Basic OS** and the manual entry for **wdLib**(1) for more information.

Message Logging

A simple message logging facility allows error or status messages to be sent to a logging task, which formats and outputs the messages to a system-wide logging device, such as the system console, disk, or accessible memory. The message logging facility can be used from interrupt level or task level.

See **4. I/O System** and the manual entry for **logLib**(1) for more information.

Memory Allocation

Vx960 supplies a UNIX source-compatible memory management facility useful for allocating, freeing, and reallocating blocks of memory from a "memory pool." The size of the pool can be set by the user. Blocks of arbitrary size can be allocated. This memory scheme is built on a much more general mechanism that allows Vx960 to manage several separate memory pools.

See the manual entry for **memLib**(1) for more information.

String Formatting and Scanning

Vx960 includes string formatting and scanning subroutines that implement UNIX-compatible *printf/scanf* format-driven encoding and decoding.

See the manual entries for fioLib(1) and stdioLib(1) for more information.

Linear and Ring Buffer Manipulations

The library bLib(1) contains buffer manipulation functions such as copying, filling, comparing, and so on, that have been optimized for speed. The library rngLib(1) provides a set of general ring buffer routines that manage first-in-first-out circular buffers. These ring buffers allow a single writer and a single reader to access a ring buffer simultaneously without being required to interlock their accesses.

Linked-List Manipulations

The library lstLib(1) contains a complete set of routines for creating and manipulating doubly-linked lists.

1.13 Board Support Packages

Two target-specific libraries, sysLib and sysALib, are included with each port of Vx960. These libraries are the heart of Vx960's portability because they provide an identical software interface to the hardware functions of all boards. They include routines for hardware initialization, interrupt handling and generation, hardware clock and timer management, mapping of local and bus memory spaces, memory sizing, and so on.

2

Getting Started

Contents

2

Getting Started

2.1 Introduction

One of Vx960's biggest strengths is the ease with which you can get your development environment up and running. If you have the necessary hardware and software components, you should be able to install, configure, and boot Vx960, and be downloading and debugging application code in less than a day.

This chapter provides step-by-step procedures for getting started with Vx960. It covers:

- Installing the Vx960 files from the distribution tape.
- Configuring the software on the UNIX host[1] system.
- Configuring the target hardware, typically a single board computer plus Ethernet hardware.
- Booting Vx960 via Ethernet using the boot ROMs.
- Downloading and running a demo program on Vx960, and exercising Vx960's testing and debugging facilities.
- Troubleshooting, if anything goes wrong while bringing up Vx960.

1. In network terminology, each node, including a Vx960 target, is a network *host*. Do not confuse this host with the cross-development *host/target* distinction in Vx960 documentation.

This chapter discusses bringing up Vx960 in a relatively simple configuration. Other chapters elaborate more advanced options. In particular, **6. Network** discusses gateways, NFS, multiprocessor target systems, and so on. Also, **8. Configuration** discusses the procedures for reconfiguring and rebuilding Vx960.

Vx960 is a flexible system that has been ported to many different hardware configurations. This chapter assumes the typical minimum target system configuration shown below in Figure 2-1 and consisting of the following:

Chassis	A card cage with backplane, typically VMEbus, and power supply. (An embedded board design may not have a backplane bus.)
CPU board	A single-board computer (*target*) that Vx960 has been ported to.
Ethernet board	An Ethernet controller board (some CPU boards include the Ethernet controller on-board).
Console	An ASCII terminal or a serial port on a workstation; this is required for initial setup only.

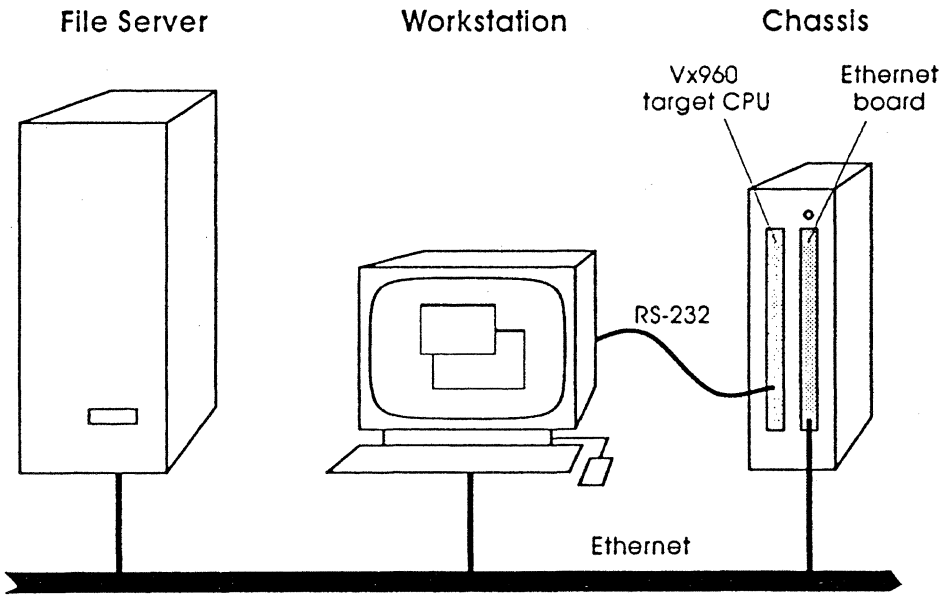


Figure 2-1. Example Minimum Configuration for Vx960

For more detailed information pertaining to your target CPU, you should also consult the target-specific information supplied for each port of Vx960.

2.2 Installing Cross-Development Tools

If you want to install the GNU/960 tool set, refer to *Installing the GNU/960 Tools* section in your GNU/960 documentation. This section provides step-by-step instructions on installing the GNU/960 tool set.

2.3 Installing Vx960

Vx960 is distributed on a tape in **tar** format.This tape contains the generic Vx960 files and all the current Board Support Package for the i960 microprocessor. To install Vx960, create a directory on your host system, for example, /usr/vx, change to that directory, and extract the system from the tape using **tar**. Typical commands to do this are:

```
% mkdir /usr/vx
% chmod 755 /usr/vx
% cd /usr/vx
% tar xvf [tape_device_of_choice]
```

Note: This manual refers to Vx960 directories and files with the pathname starting at /usr/vx... . However, Vx960 does not assume or require this pathname.

Refer to the *Vx960 Release Notes* or see your system administrator for the commands specific to your host. In the /usr/vx directory there is a file called README. Consult this file for the most current information about the version of Vx960 that you are installing. Run the shell script /bin/vxInstall.sh to install Vx960.

Your vx directory has a number of subdirectories and files in it. The directory /usr/vx/config contains a directory for each of the target types you have ordered. In each of those directories you find the system boot image and its symbol table, i.e.:

/usr/vx/config/*target*/vxWorks	system boot image
/usr/vx/config/*target*/vxWorks.sym	system symbol table

These are the only files you need to get started with Vx960.

For access to Vx960 tools that run on the host, you may also want to put the appropriate bin directory in your UNIX shell search path. If you are using a Sun system, the tool arch is useful for this. For example, you might add the following to your .cshrc file:

```
set arch='arch'
setenv PATH .:/usr/vx/bin/$arch:...
```

The Vx960 directory tree and file contents are described in A. Directories and Files.

2.4 Configuring the Host System

Before you can boot the Vx960 from the host, you must configure the network software on the UNIX host. In configuring the host network software, you must:

- Initialize the host network software.
- Establish the Vx960 system name and network address on the host.
- Give the Vx960 system appropriate access permissions on the host.

The following sections describe the procedures in detail. Some of these procedures may require root permissions. Also, some UNIX systems may require different procedures. Consult your system administrator.

2.4.1 Hosts Supported

The following is a list of the hosts supported by Vx960. If the operating system for your host is upwardly compatible, the existing GNU/960 tools may be supported.

The source for all Vx960 binaries are provided to allow other hosts to be supported. If the GNU/960 tools are supported, Vx960 supports that host.

Processor	Operating System
Sun-4	SunOs4.0.3c
Sun-3	SunOs4.0.3c
Sun-386i	SunOs4.0.1
VAX 8600	Ultrix-32 V3.1 (Rev.9)
HP9000/300	HP/UX V7.0
Compaq Deskpro 386/33	Intel System V R3.2
DECStation 3100	ULTRIX V2.0 (Rev 7)
IBM RS/6000	AIX Version 3.1
HP/Apollo Series 400	Domain/OS SR 10.3
i386 systems	System V R3.2

2.4.2 Initializing the Host Network Software

You cannot initialize most UNIX systems without going through a process of executing multiple startup Scripts. For additional information, consult your UNIX system administration manuals if network initialization needs to be added to your UNIX startup procedure.

2.4.3 Establishing the Vx960 System Name and Address: /etc/hosts

The UNIX host system maintains a database of the names and network addresses of systems accessible from the local system. This database is kept in the ASCII file /etc/hosts which contains a line for each system. Each line consists of an Internet address and the name(s) of the system at that address. This file must have entries for your host UNIX system and the Vx960 target system.

For example, suppose your UNIX host system is called "mars" and has been assigned Internet address 90.0.0.1, and you want to name your Vx960 target "phobos" and assign it address 90.0.0.50. The file /etc/hosts should then contain the following lines:

```
90.0.0.1        mars
90.0.0.50       phobos
```

Note: If your system is running the Network Information Service (NIS), formerly the Yellow Pages service, then the hosts database is maintained by NIS facilities that are beyond the scope of this introduction. Consult your UNIX system administration manuals if you are running NIS.

2.4.4 Giving Vx960 Access to the Host: .rhosts and /etc/hosts.equiv

The UNIX system restricts network access via remote login, remote command execution, and remote file access. This is done with the .rhosts file in the user's home directory, and more globally with the /etc/hosts.equiv file. The .rhosts file contains a list of system names that have access to your login. Thus, in our example, to allow your Vx960 system to log in with your user name and access files with your permissions, you would create a .rhosts file in your home directory containing the line:

```
phobos
```

The /etc/hosts.equiv file provides a less selective mechanism. Systems listed in this file are allowed login access to any user defined on the local system (except the super-user root). Thus, adding the Vx960 system name to /etc/hosts.equiv allows the Vx960 system to log in with any user name on the system.

2.5 Configuring the Target Hardware

Configuring the target hardware involves the following tasks:

- Installing the Vx960 boot ROMs.
- Jumpering the target CPU and Ethernet boards.
- Installing the boards in the backplane.
- Connecting the serial and Ethernet cables.

Use the following general procedures, as appropriate to your particular target system hardware. Refer to the specific information in the documentation supplement for your target.

2.5.1 Installing the Boot ROMs

Intel supplies boot ROMs for your selected targets as part of the Vx960 delivery. Use these ROMs to boot Vx960 over the network. Install the appropriate set of boot ROMs on your target board(s).

When installing these devices, observe the following:

- Install each device only in the socket indicated on the label.
- Note the correct orientation of pin 1 for each device.
- Use anti-static precautions whenever working with integrated circuit devices.

The file **bootrom.hex** exists or can be made in each target directory. This file contains the boot ROMs in Intel Hex format, and is suitable for downloading to a PROM programmer for making your own Vx960 boot ROMs. To make **bootrom.hex**, change to the **vx/config/***target* directory for your target, and type:

```
% make bootrom.hex
```

Note: On targets with more than one boot ROM, this command produces an equal number of boot ROM files. See the appropriate target Board Support Package document for more information.

2.5.2 Setting the Board Jumpers

Many CPU and Ethernet controller boards have configuration options that are selected by hardware jumpers. These jumpers must be installed correctly to bring up Vx960. You can determine the correct jumper configuration for your target CPU from the information provided in the documentation supplement for your target or use the **jump** program supplied with the Vx960 system. The **jump** program runs on UNIX and uses simple ASCII graphics to display jumpering diagrams for CPU and Ethernet controller boards.

To run **jump**, type:

```
% jump /usr/vx/config/target
```

where *target* is the Vx960 directory name which corresponds to your target system. The program prompts you with a selection of CPU boards for which jumpering diagrams can be displayed. This invocation of jump assumes you have included the appropriate Vx960 bin directory in your UNIX shell search path as described in **2.2 Installing Vx960.**

2.5.3 Installing the Boards in the Backplane

To assemble your target hardware, first install the CPU board in slot 1 of the backplane. If you are using a separate Ethernet controller board, install it in slot 2 of the backplane.

If you are using a VMEbus backplane, consider the following:

- P1 and P2 Connectors

  The P1 connector must be bussed across all the boards in the system. Many systems also require the P2 bus. Some boards require power on the P2 connector. Also some configurations require the extended address and data lines of the B row of the P2 bus.

- System Controller

  The VMEbus requires a "system controller" to be present in slot 1. Many CPU boards have a system controller on board that can be enabled or disabled by hardware jumpers. On such boards, the system controller should be enabled in slot 1 and disabled in all others. The diagrams shown by the **jump** program (described above) indicate the location of the system controller enable jumper, where available. Alternatively, a separate system controller board can be installed in slot 1, and the CPU and Ethernet boards can be plugged into the next two slots.

- Empty Slots

  The VMEbus has several daisy chained signals that must be propagated to all the boards on the backplane. If you leave any slot empty between boards on the backplane, you must jumper the backplane to propagate the BUS GRANT and INT ACK daisy chains.

2.5.4 Connecting the Cables

All Vx960 supported target CPUs include at least one on-board serial port. This serial port must be connected to an ASCII terminal, at least for the initial configuration of the boot ROMs and getting started with Vx960. Subsequently, Vx960 can be configured to boot automatically without a terminal. Refer to the CPU board hardware documentation for proper connection of the RS-232 signals.

For the Ethernet connection, a transceiver cable must be connected from the Ethernet controller to an Ethernet transceiver.

2.6 Booting Vx960

If you have correctly configured your host software and target hardware, you are ready to turn on the target system power and to boot Vx960. When you power-on and reset the target hardware, the target system terminal or workstation serial port is talking to the Vx960 boot ROMs. The boot ROMs first display a banner page and then start a 7-second countdown, visible on the screen as shown in Figure 2-2. Unless you press any key on the keyboard within that 7 seconds, Vx960 automatically boots with a default configuration.

Power-on, reset the target, and stop the automatic boot by pressing any key on the keyboard. The boot ROMs display the Vx960 boot prompt:

```
[Vx960 Boot]:
```

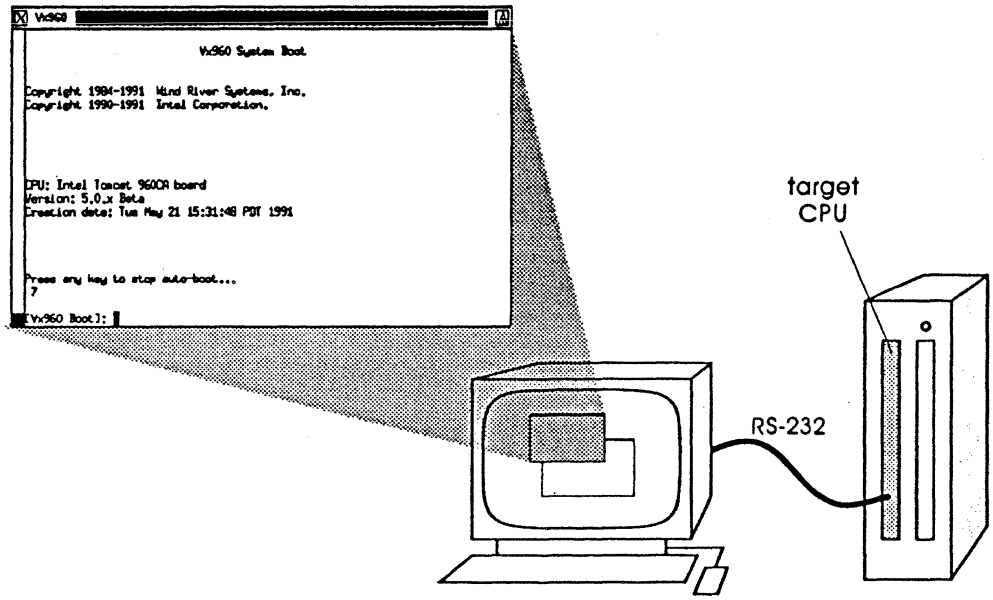


Figure 2-2. Boot ROMs Communication and Initial Display

2.6.1 Boot ROM Commands

The Vx960 boot ROMs provide a limited set of commands. To see a list of available commands, type the help command **h** or **?** followed by a RETURN:

```
[Vx960 Boot]: ?
```

Table 2-1 lists the Vx960 boot commands and parameters, with a brief description of each.

2.6.2 Entering Boot Parameters

Before booting Vx960, set the configuration parameters for the boot, including the host and target network addresses, the file to be booted, the user name, etc. To display the current boot parameters, type:

```
[Vx960 Boot]: p
```

Table 2-1. Vx960 Boot Commands

Command	Description
h	"Help" command – print a list of available boot commands.
?	Same as **h**.
@	Boot (load and execute the file) using the current boot parameters.
p	Print the current boot parameter values.
c	Change the boot parameter values.
l	Load the file using current boot parameters, but without executing.
g *adrs*	Go (execute) at hex address *adrs*.
d *adrs*[**,** *n*]	Display *n* words of memory starting at hex address *adrs*. If *n* is omitted, the default is 64.
m *adrs*	Modify memory at location *adrs* (hex). The system prompts for modifications to memory, starting at the specified address. It prints each address, and the current 16-bit value at that address, in turn. You can respond in one of several ways: RETURN – Do not change that address, but continue prompting at the next address. *number* – Set the 16-bit contents to *number*. . (dot) – Do not change that address and quit.
f *adrs*, *nbytes*, *value*	Fill *nbytes* of memory starting at *adrs* with *value*.
t *adrs1*, *adrs2*, *nbytes*	Copy *nbytes* of memory starting at *adrs1* to *adrs2*.
s [**0** | **1**]	Turn the CPU system controller ON (**1**) or OFF (**0**) (only on boards that have software enable of the system controller).
e	Display a synopsis of the last occurring Vx960 exception.
n *netif*	Display the ethernet address of the network interface device *netif*.

You see a display similar to the following, which corresponds to the example configuration shown in Figure 2-3. (The **p** command does not display blank fields as in this illustration.)

```
boot device            : ei
processor number       : 0
host name              : mars
file name              : /usr/vx/config/hkv960/vxWorks
inet on ethernet (e)   : 90.0.0.50
inet on backplane (b)  :
host inet (h)          : 90.0.0.1
gateway inet (g)       :
user (u)               : fred
ftp password (pw)(blank=use rsh) :
flags (f)              : 0
target name (tn)       : phobos
startup script (s)     : /usr/vx/config/hkv960/startup.cmd
other (o)              :
```

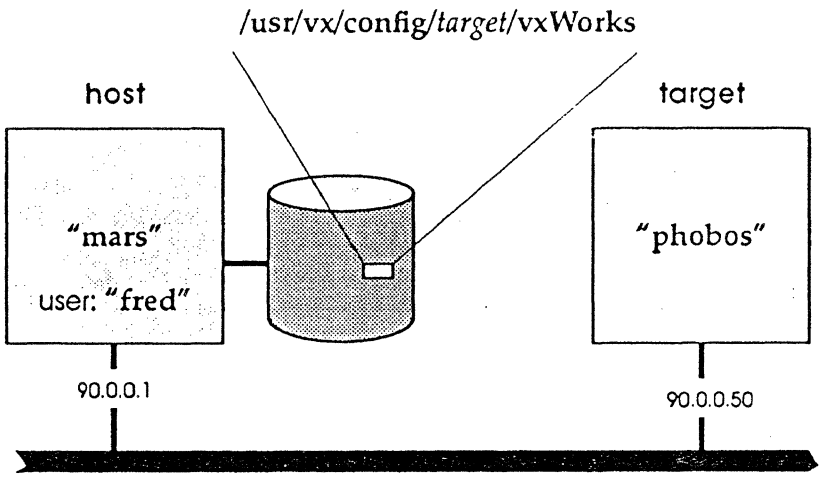


Figure 2-3. Boot Configuration Example

The meaning of each of these parameters is described in the next section.

To change the boot parameters type:

```
[Vx960 Boot]: c
```

You are prompted for each parameter. If the contents of a particular field need not be changed, press RETURN. If a field is to be cleared, type a period (.), followed by RETURN.

Network information *must* be entered to match your particular system configuration. The Internet addresses should correspond to those in /etc/hosts on your UNIX host, as described previously.

If your target board has non-volatile RAM, the boot parameters are stored there and retained even while the system power is turned off. For subsequent power-ons or system resets, the boot ROMs use the stored parameters for the automatic boot configuration.

2.6.3 Description of Boot Information

Each of the boot parameters is described below. The letters in parentheses indicate how the parameters can be specified in the command-line boot procedure described in **2.6.5 Alternative Booting Procedures**.

`boot device` — The type of network device Vx960 boots from. This must be one of the network drivers included in the boot ROMs (e.g., "ei" for an Intel 82596 LAN Coprocessor). Due to limited space in the boot ROMs, only a few drivers can be included. A list of included drivers is displayed at the bottom of the help screen (type `?` or `h`).

`host name` — The name of the host machine to boot from; "mars" in our example. This is the name by which the host is known to Vx960; it need not be the same name used internally by the host.

`boot file` — The full pathname of the Vx960 object module to be booted (/usr/vx/config/*target*/vxWorks in the example).

`inet on Ethernet (e)` — The Internet address of a target system with an Ethernet interface (90.0.0.50 in the example).

`inet on backplane (b)` — The Internet address of a target system with a backplane interface (this should be blank in our example).

`host inet (h)` — The Internet address of the UNIX host to boot from (90.0.0.1 in our example).

gateway inet (g) — The Internet address of a gateway node if the host is not on the same network as the target (this should be blank in our example).

user (u) — The user name with which Vx960 accesses the UNIX host; the named user should have read access to the Vx960 file being booted. Vx960 must have access to this user's account, either via the files **.rhosts** or **/etc/hosts.equiv** discussed above, or by the *ftp* password provided below.

ftp password (pw) — The "user" password. This field is optional. If a password is given, *ftp* is used instead of *rsh*. If you do not want to use *ftp*, leave this field blank.

processor number — A unique identifier for the target in systems with multiple targets on a backplane (this should be zero in our example). For an embedded control system, the number is always zero.

flags (f) — Configuration options specified as a numeric value that is the sum of the values of selected option bits defined below.

- **0x01** = Do not enable the system controller, even if the processor number is 0. (This option is board specific; refer to your target documentation.)
- **0x02** = Load all Vx960 symbols, instead of just globals.
- **0x04** = Do not auto-boot.
- **0x08** = Auto-boot fast (short countdown).
- **0x20** = Disable login security.

target name (tn) — The name of the target system to be added to the host table; "phobos" in our example.

startup script (s) — The complete path name of the startup script to execute after the system boots. If omitted, no script is executed. To enable this feature, INCLUDE_STARTUP_SCRIPT must be defined in **configAll.h**. For more information on the use of startup scripts, see **9.4.3 Scripts**.

other (o) — This parameter is unused and available for applications.

2.6.4 Booting

Once you have entered the boot parameters, initiate booting by typing:

```
[Vx960 Boot]: @
```

The Vx960 boot ROMs print the boot parameters and the download begins. While Vx960 is booting, you see the size of each Vx960 segment as it is loaded. Once the system is loaded, the boot ROMs display the entry address and transfer control to the loaded Vx960 system. When Vx960 has completed initialization, it displays a banner page like the one shown in Figure 2-4, followed by the Vx960 shell prompt "->".

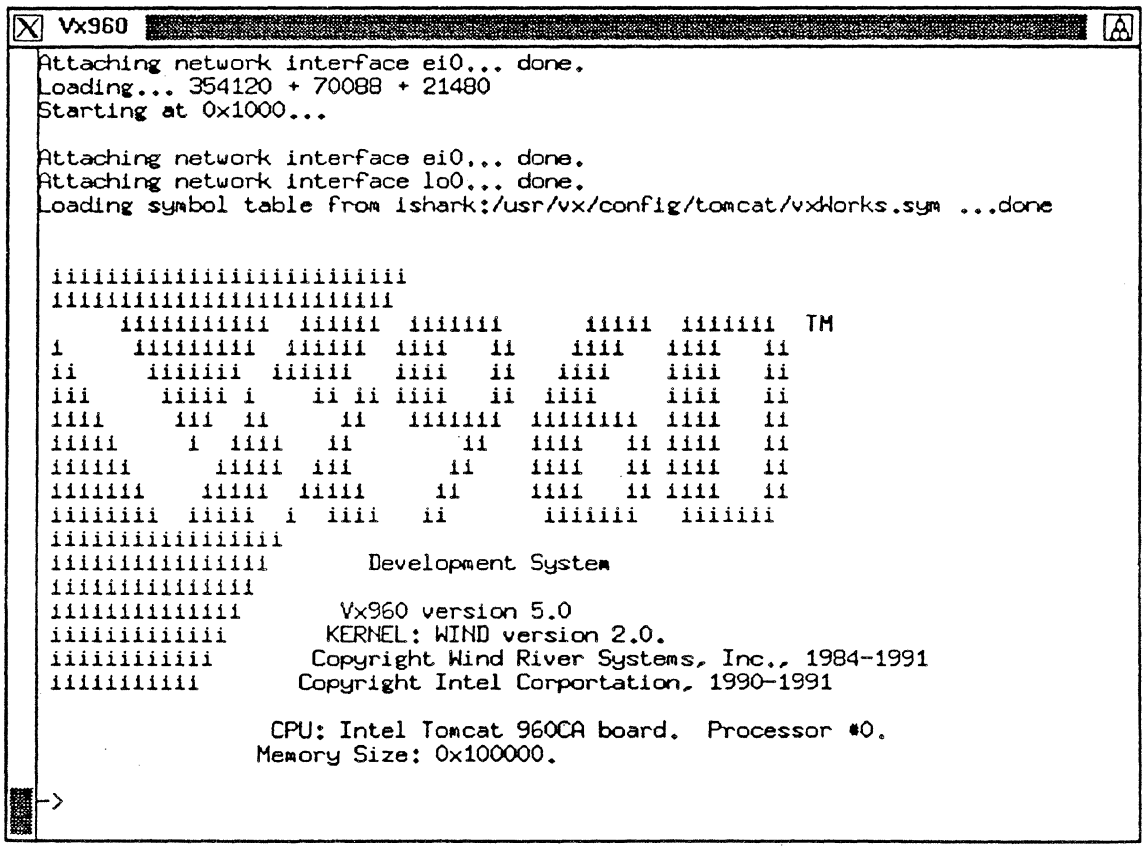


Figure 2-4. Vx960 Booting Display and Banner

2.6.5 Alternative Booting Procedures

2.6.5.1 Command-Line Parameters

You can supply the boot ROMs with all the parameters on a single command line at the boot prompt (`[Vx960 Boot]:`). For example:

```
$ ei(0,0)mars:/usr/vx/config/hkv960/vxWorks e=90.0.0.50 h=90.0.0.1 u=fred
```

The order of the assigned fields (containing "=" signs) is not important. Assigned fields that are irrelevant should be omitted. The codes for the assigned fields correspond to the letter codes shown in parentheses when you typed the `p` command to display the boot parameters. For a complete description of the format, see the manual entry for *bootStringToStruct*( ) in bootLib(1).

This method can be useful if you have programmable function keys. You can program a function key with a command line appropriate to your configuration.

2.6.5.2 Non-Volatile RAM

As noted previously, if your target CPU has non-volatile RAM, all the values you enter in the booting parameters are retained in the non-volatile RAM. In this case, you can let the boot ROMs auto-boot without even having a terminal connected to the target system.

2.6.5.3 Reprogramming Boot ROMs

You can burn your boot ROMs with the correct boot parameter values. With this method, you no longer need to alter the boot parameters when Vx960 attempts to boot. The default boot parameters are specified in the file config.h in the configuration directory for your target CPU (/usr/vx/config/*target*/config.h). This file contains a *#define* of DEFAULT_BOOT_LINE. The value of this constant can be changed to the appropriate boot parameters for your system. After editing config.h, type the following in the configuration directory for your target:

```
% make bootrom.hex
```

The file **bootrom.hex** contains the boot ROMs in standard Intel Hex format and is ready to download to an EPROM programmer. For the sake of thoroughness, type "make" so that any other files affected by your changes are remade.

2.6.5.4 Using Alternate ROMs

When the target system is initialized, it starts executing from the boot ROMs. Boot ROMs can be built three ways: compressed, normal, or ROM resident. A compressed boot ROM has a smaller image but takes longer to boot because it must uncompress the data from ROM to RAM. A normal ROM will boot faster, but takes up more space in the ROM. The normal method was the method supported in the 4.0.2 release of Vx960. When the boot ROMs are built ROM resident, only the data section is copied from the ROM to RAM at initialization time. The actual code will run from ROM. This will probably execute slower than the same code executing from RAM. Using the ROM resident option allows for the fastest boot. The file **/usr/vw/config/all/bootInit.c** can be compiled three different ways. The -DROM_RESIDENT, -DUNCOMPRESS and flags create ROM_RESIDENT and normal code respectively. The default is to create compressed code.

See /usr/vw/config/<target>/Makefile or /usr/vw/config/all/bootInit.c for more information.

2.6.6 Rebooting

When Vx960 is running, there are several way you can reboot Vx960:

- Press the reset button on the target system.
- Type a ^X at the target console terminal.
- Invoke *reboot( )* from the Vx960 shell. This can be done even when remotely logged in to Vx960.

When Vx960 is rebooted in any of these ways, the auto-boot sequence begins again from the countdown.

2.6.7 Reconfiguring and Rebuilding Vx960

The Vx960 image supplied on the distribution tape works with all standard configurations of your supported targets. You may find that you want to reconfigure the distributed Vx960 to change a parameter or to include or exclude optional modules.

Many Vx960 parameters are specified in the two configuration headers:

- /usr/vx/config/all/configAll.h
- /usr/vx/config/*target*/config.h

You can modify these headers to suit your configuration. Then change to the /usr/vx/config/*target* directory and type:

```
% make
```

This rebuilds the system image vxWorks and the system symbol table vxWorks.sym. Reconfiguration of Vx960 is discussed in detail in **8. Configuration.**

2.7 Playing With Vx960 and a Demo Program

This section introduces you to program development and debugging using Vx960. The instructions in this section describe how to use the shell to load and run a very simple program on the Vx960 system you just booted.

2.7.1 The Vx960 Shell

When Vx960 comes up, you are interacting with the Vx960 shell. The Vx960 shell is an interactive C-expression interpreter. The shell reads lines of input from the terminal, parses and evaluates each line, and writes the result of the evaluation to an output stream. The shell accepts the same expression syntax as the C compiler with a few variations. This simple mechanism can be used in many different ways.

The sections below provide a brief introduction to using the Vx960 shell. For a more detailed discussion, see **9. Shell.**

2.7.1.1 Special Characters

As you type in commands to the shell, you can use the following special characters:

Table 2-2. Vx960 Shell Special Characters

Command	Description
^H	Delete a character (backspace).
^U	Delete an entire line.
^C	Abort and restart the shell.
^X	Reboot (trap to the ROM monitor).
^S	Temporarily suspend output.
^Q	Resume output.
ESC	Toggle between input mode and edit mode.

2.7.1.2 Examples of Shell Usage

The following examples illustrate some typical uses of the shell. The shell prompt for interactive input is "->". User input is shown in **bold face** and shell responses are shown in regular face.

```
-> 68
value = 68 = 0x44 = 'D'

-> x = 13
new symbol "x" added to symbol table
x address = ...:  value = 13

-> printf ("hello world, x = %d\n", x)
hello world, x = 13
value = ...
```

2.7.1.3 Shell History and Line Editing

The Vx960 shell keeps a list of the commands that were typed in previously. To see this list, type the "history" command:

```
-> h
```

A list of the last 20 shell commands is displayed on the screen.

The Vx960 shell provides a history mechanism similar to the UNIX K-Shell, which allows you to recall lines from the shell history, edit them, and re-execute them. Typing the ESC key changes the shell from input mode to editing mode.

In editing mode, you can change previous command lines using UNIX **vi**-like commands. The **k** command recalls successive previous commands from the history list, and the **j** command moves you forward in the history list. The **h** and **l** commands move the cursor left and right; an **x** deletes a character; etc. A RETURN sends the entire command to the shell regardless of cursor position or input mode. The Vx960 shell history and editing commands are described in detail in **9. Shell**.

2.7.1.4 Remote Access to the Shell

In addition to accessing the Vx960 shell on the target console terminal, you can also access the shell from a terminal or workstation on your UNIX host using the remote login facility **rlogin**. To log in to our example Vx960 target system, you would type the following on your UNIX terminal:

```
% rlogin phobos
```

The **telnet** remote login protocol is also available. See **9. Shell** for more information.

2.7.2 On-Line Help

Two sources of on-line help are provided: one from the host UNIX system and the other from Vx960 directly.

2.7.2.1 From Vx960

The following help commands are available if the corresponding subsystem has been included in your Vx960 system configuration:

- *help*( )
- *dbgHelp*( )
- *timexHelp*( )

- *spyHelp( )*
- *netHelp( )*
- *nfsHelp( )*

These help commands display brief summaries of various Vx960 subsystems. They are not intended to be tutorial, but rather are convenient memory joggers for some commonly used Vx960 functions.

2.7.2.2 From the Host: vxman

All Vx960 reference documentation is distributed for on-line use from the UNIX host using vxman, which is analogous to the UNIX man command. The vxman facility lets you to display the manual page for any Vx960 library, subroutine, driver, or tool. For example, to display the manual entry for *taskSpawn( )*, type:

```
% vxman taskSpawn
```

To include vxman in your UNIX shell search path, follow the procedures suggested in **2.2 Installing Vx960**. For more information on the use of this facility, see the manual entry for vxman, or type:

```
% vxman vxman
```

2.7.3 A Demo Program

To illustrate some basic Vx960 concepts and techniques, this section describes loading and debugging a demonstration program called demo.c. This program prints its own task ID, task name, and start-up parameter on the console, and then exits. Additional demo programs are provided in the /usr/vx/demo directory.

In this example, the following assumptions are made:

- The Vx960 tape was loaded into /usr/vx.
- Your home directory is /usr/fred.
- You are playing with the demo program in the directory /usr/fred/vxdemo.
- The host machine's name is mars.

To begin, create a working directory somewhere on your host system (/usr/fred/vxdemo for this example) and copy /usr/vx/demo/1/demo.c into it:

```
% cd /usr/fred
% mkdir vxdemo
% cd vxdemo
% cp /usr/vx/demo/1/demo.c demo.c
```

Compile demo with the following command:

```
% gcc960 -c -ACA -O -I/usr/vx/h demo.c
```

The **-c** flag suppresses linking with the GNU/960 C libraries and leaves the undefined externals unresolved. These are resolved by the Vx960 linking loader. The **-O** flag optimizes the code (this flag is optional). The **-I** flag tells the compiler where to find the Vx960 header files that are included in demo.c. You need to include architecture-specific flags; for example, if you are cross-compiling for a 80960CA processor, you need to include the -ACA flag.

If you are on a "big-endian" architecture host such as a Sun-4 or HP 9000, you also need to swap the byte order of the object file. With the GNU/960 tool set, this is accomplished with a command of the following form:

```
% objcopy -l demo.o
```

The **-l** flag specifies that the named files are to be byte-swapped to "little-endian" format, which is the Vx960 native byte order. On little-endian hosts, such as a Compaq, this step isn't necessary, but performing this step on such hosts is harmless and the resulting object loads without problems.

If Vx960 has not yet been booted, now is the time to do so. Then access Vx960 either by its system console or by remote login using **rlogin**.

Next check the user name that the Vx960 system uses on the remote host system with the *whoami*( ) command:

```
-> whoami
```

The user name is used for access privileges. When Vx960 first comes up, the user name is set to the user name specified in the boot parameters. If the current user name is not yours, change it with the *iam*( ) command:

```
-> iam "fred"
```

Next set your current working directory with the *cd*( ) command:

```
-> cd "mars:/usr/fred/vxdemo"
```

(where mars is the name of the host system as specified in the boot parameters). This sets the default directory to be the directory where you compiled **demo.c**.

The command to load applications into Vx960 system memory is *ld*( ), which performs three functions:

- Loads the program into memory.
- Adds the program's symbols to the system symbol table.
- Resolves the program's external references.

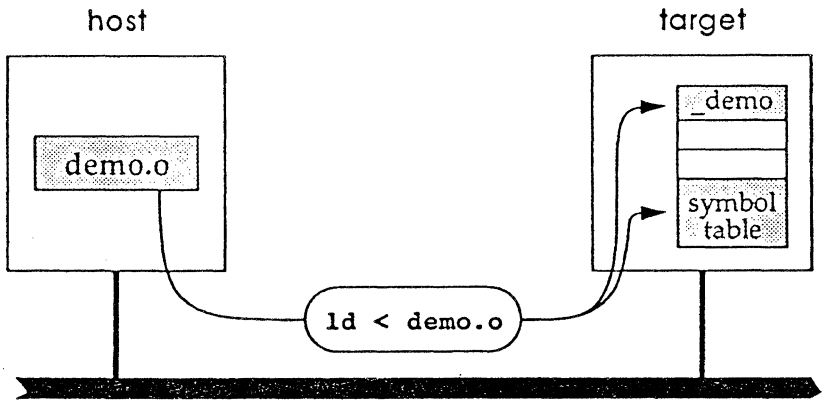


Figure 2-5. Downloading a Program

To load the demo program, type:

```
-> ld < demo.o
value = 0 ...
```

When loaded, all the program's undefined externals are resolved by the Vx960 linking loader. You can see this by disassembling the routine. Type:

```
-> l demo
-> l
-> l
```

The *l*( ) command shows a symbolic disassembly of the code in demo.c. The assembler code shows addresses symbolically. Any code in the system can be disassembled (try *l*( ) with *printf*( ), for example). By default, *l*( ) disassembles ten instructions at a time; subsequent calls continue from the previous address.

Try the following commands:

```
-> demo 1234                                                    ❶
-> sp demo, 1234                                                ❷
-> taskSpawn "tdemo", 100, 0, 2000, demo, 1234                  ❸
```

Each of these runs the demo program, but in a different way. Example ❶ runs demo as a subroutine in the shell's context and tells you that its task name is *tShell*. ❷ spawns demo as a separate task, and assigns it a default task name and a default priority. The default task name is a number prefixed with a "t". ❸ also spawns demo, but with explicit parameters for the task name (*tdemo*), priority (100), and stack size (2000).

The convention of using a "t" to prefix default task names and the names of system tasks avoids name conflicts with symbols in the system symbol table. For more information, see **9.3.3.6 Task References.**

Type:

```
-> i

NAME         ENTRY      TID    PRI STATUS   PC      SP     ERRNO  DELAY
---------- --------- ------- --- ------ ------ ------ ------- -----
tExcTask   _excTask   3fbe04   0 PEND   15dee  3fbd48      0      0
tLogTask   _logTask   3fa8c8   0 PEND   15dee  3fa804      0      0
tShell     _shell     3c60f8   1 READY  2b590  3c5dcc      0      0
tRlogind   _rlogind   3d7084   2 PEND   4b374  3d6ea4      0      0
tNetTask   _netTask   3f5298  50 PEND   4b374  3f5240      0      0
```

The *i*( ) command shows the tasks currently running. The tasks that you see – *tShell*, *tLogTask*, *tExcTask*, *tNetTask*, and *tRlogind* – are system tasks that are present.

Now try:

```
-> period 5, demo, 5
-> i
```

The *period*( ) command spawns *periodRun*( ) as a new task which calls demo in its context every 5 seconds. Now demo should be printing its message every five seconds. The next *i*( ) command again shows all the tasks currently running. Notice the new task whose entry point is *_periodRun*:

```
t2          _periodRun 3c3894 100 DELAY   433d0 3c3858        0   136
```

You can insert a breakpoint with the *b*( ) command:

```
-> b printf
```

This sets a breakpoint at the address of *printf*( ). The *_periodRun* task should hit it within 5 seconds. When it does, a message prints out telling you about it:

```
Break at 0x00b21a: _printf                  Task: 0x3c3894 (t2)
```

Now type:

```
-> i

t2          _periodRun 3c3894 100 SUSPEND  b21a 3c3828        0     0
```

The information that *i*( ) prints now shows that *_periodRun* is suspended (because it hit a breakpoint).

Then type:

```
-> tt

 547a0 _vxTaskEntry +10 : _periodRun ( )
 2f4a4 _periodRun   +24 : _demo ( )
ffee60 _demo        +24 : _printf (feec0, c4170, c9130, 5, 0, )
```

The task trace command *tt*( ) shows where the task was when it hit the breakpoint. You can see which routines were called by other routines, and you can see what parameters were used in the call to the routine at which the breakpoint occurred.

To continue, type:

```
-> bdall
-> c
```

The *bdall*( ) command deletes all breakpoints, and the *c*( ) command continues the task. If the breakpoint had not been deleted, the task would have hit it again in 5 seconds.

Now type:

```
-> period 7, demo, 7                                    ❶
-> i                                                    ❷
-> sp demo, 1111                                        ❸
```

When you type ❶, you spawn another task, whose entry point is also *_periodRun* (in fact, it is the same code in memory), which calls **demo** every 7 seconds. The information display ❷ shows another task, with a different ID and task name but the same entry point. Since **demo** is called from the context of a different task, the 7-second cycle gives you an ID different from the 5-second cycle which is still running. In fact, when you spawn **demo** as a task again ❸, it tells you another ID, even though it is the same code in memory. Remember, there is only one copy of the **demo** code in memory, but it is being executed in three different task contexts.

To delete the periodic *_periodRun* tasks, use the task-delete command *td*( ). The argument to *td*( ) is the task ID, which you can find using *i*( ). Thus, our 5-second example, whose task whose name is *t2* and whose ID is 3c3894, can be deleted by typing:

```
-> td 0x3c3894
```

Most Vx960 routines that take a task parameter require a task ID. However, specifying a task ID can be cumbersome since the ID is an arbitrary and possibly lengthy number.

To accommodate interactive use, Vx960 shell expressions can reference a task by either task ID or task name. The Vx960 shell attempts to resolve a task argument to a task ID; if no match is found in the system symbol table, it searches for the argument in the list of active tasks. When it finds a match, it substitutes the task name with its matching task ID.

Thus, a simpler way of deleting the task in the previous example is:

```
-> td t2
```

2.8 Troubleshooting

If you encountered problems booting or exercising Vx960, there are many possible causes. This section discusses the most common sources of error and how to narrow

the possibilities. Please read **2.8.1 Things to Check** below before contacting your technical support group. Often, by rechecking the installation steps, your hardware configuration, and so forth, you can locate the problem.

2.8.1 Things to Check

Most often, a problem with running Vx960 can be traced to hardware or software configuration errors. Consult the following checklist to locate a problem.

2.8.1.1 Hardware Configuration

- **Limit the number of variables.** Start with a minimal configuration of a single target CPU board and an Ethernet board.
- **Be sure your backplane is powered and bussed.** For the VMEbus backplane, most configurations require that the P2 B-Row be bussed and that there is power supplied to both the P1 and P2 connectors.
- **Be sure boards are in adjacent slots.** The only exception to this is if the backplane is jumpered to propagate the BUS GRANT and INT ACK daisy chains.
- **Check that the RS-232 cables are correctly constructed.** In most cases, the documentation accompanying the hardware explains the cabling requirements for that hardware.
- **Check the boot ROMs for correct insertion.** If the CPU board seems dead when applying power (some have front panel LEDs), the boot ROMs can not be correctly inserted. You can also validate the checksum printed on the boot ROM labels.
- **Press the RESET button if required.** Some system controller boards do not reset the backplane on power-on, requiring you to reset manually.
- **Make sure all boards are jumpered properly.** Use the **jump** program described in this chapter to determine the correct jumpering of your target and Ethernet boards.

2.8.1.2 Booting Problems

- **Check the Ethernet transceiver site.** For example, connect a known working system to the transceiver and check whether the network functions.
- **Verify Internet addresses.** An Internet address consists of a network number and a host number. There are several different classes of Internet addresses that assign different parts of the 32-bit Internet address to these two parts, but in all cases the network number is the most significant bits and the host number is least significant bits. The simple configuration described in this chapter assumes that the host and target are on the same network, i.e., have the same network number. (See **6. Network** for a discussion of setting up gateways if the host and target are not on the same network.) If the target Internet address is not on the same network as the host, the Vx960 boot ROMs say:

  ```
  Error loading file: status = 0x33.
  ```

  0x33 corresponds to host *errno* 51 (decimal) ENETUNREACH.

  If the target Internet address is not in **/etc/hosts**, then the host does not know about your target. The Vx960 boot ROMs receive an error message from the host:

  ```
  host name for your address unknown
  Error loading file: status = 0x320001.
  ```

  The Vx960 module number for **hostLib** 50 (decimal) is 0x32. The digit "1" corresponds to S_hostLib_UNKNOWN_HOST.

- **Verify host file permissions.** The target name must be listed in either *user-home-dir*/**.rhosts** or **/etc/hosts.equiv** (the target user-name can be any user on the host). Note that the user name "root" is special: having the target name in /etc/hosts.equiv is not sufficient for root access; the target name must appear in **/.rhosts** for root access.

  Make sure that the user name you are using on the target has access to the host files. To verify that the user name has permission to read the vxWorks file, try logging in on the host with the target user name and accessing the file (for instance, with the UNIX **size** command). This is what the target does when it boots.

  If you have trouble with access permissions, you might try using *ftp* (File Transfer Protocol) instead of relying on *rsh* (remote shell). If no password is specified in the boot parameters, the Vx960 object module is loaded using the *rsh* service.

However, if a password is specified, *ftp* is used. Sometimes *ftp* is easier because you specify the password instead of relying on the configuration files on the host. Also some non-UNIX systems do not support *rsh*, in which case you must use *ftp*.

- **Check host account .cshrc file.** If NFS has not been included, the Vx960 symbol table is downloaded using **rcmd**, which automatically executes the **.cshrc** of your host user account. If the processing of your .cshrc is set up to generate any standard output, it can interfere with the loading of the symbol table.

  To check whether the .cshrc file is causing booting problems, rename it temporarily and try booting Vx960 again. If this proves to be the source of the problem, you may want to set up your .cshrc file to execute conditionally any commands that generate standard output. For example, commands used to set up interactive C-shells could be grouped at the end of the .cshrc file and preceded with the following line:

```
# skip remaining setup if a non-interactive shell:
if (${?USER} == 0 || ${?prompt} == 0 || ${?TERM} == 0) exit
```

- **Helpful Troubleshooting Tools.** In tracking down configuration problems, the following UNIX tools can be helpful (you can also see your system administrator or refer to the UNIX man pages for additional information on these commands):

  netstat — This command gives various network status reports. A **-r** option displays the network routing tables. This is useful when gateways are used to access the target. The **-s** option tells you the names of the interfaces so that you can use **ifconfig**.

  ifconfig — This command reports the configuration of the specified network *interface* (e.g., **ie0** or **le0** on a SUN system). It should report that the interface is configured for the appropriate Internet address and that the interface is up.

  arp -a — This command displays the address resolution protocol tables that map Internet addresses to Ethernet addresses. Your target machine is listed if at least one packet was transferred from your target to your host.

etherfind This command can be used on many UNIX systems to watch all traffic on a network. You may need root privileges to run this command.

ping This command can be used on UNIX systems to determine whether Vx960 is up and responding up to the IP/ICMP protocol levels. You can also use **ping** to send packets and then determine if any of the packets were lost. Lost packets might indicate that there is a cable or hardware problem.

3

Basic OS

Contents

3

Basic OS

3.1 Introduction

Modern real-time systems are based on the complementary concepts of multitasking and intertask communications. A multitasking environment allows a real-time application to be constructed as a set of independent tasks, each with its own thread of execution and set of system resources. The intertask communication facilities allow these tasks to synchronize and communicate in order to coordinate their activity. In Vx960, the intertask communication facilities range from fast semaphores to message queues and UNIX-like pipes to network-transparent sockets.

Another key facility in real-time systems is hardware interrupt handling, since interrupts are the usual mechanism used to inform a system of external events. To get the fastest possible response to interrupts, interrupt handling code in Vx960 runs in a special context of its own, outside of the context of any task.

This chapter discusses the multitasking kernel, tasking facilities, intertask communication, and interrupt handling facilities, which are at the heart of the Vx960 runtime environment.

3.2 Tasks

3.2.1 Multitasking

Multitasking provides the fundamental mechanism for an application to control and react to multiple, discrete real-world events. The basic multitasking environment is provided by the Vx960 real-time kernel. Multitasking creates the appearance of many programs executing concurrently when, in fact, the kernel interleaves their execution on the basis of a scheduling algorithm. Each apparently independent program is called a *task*. Each task has its own *context*, which is the CPU environment and system resources the task sees each time it is scheduled to run by the kernel. A task's context includes:

- a thread of execution, i.e., the task's program counter
- the CPU registers and (optionally) floating-point registers
- a stack for dynamic variables and function calls
- I/O assignments for standard input, output, and error
- a delay timer
- a timeslice timer
- kernel control structures
- signal handlers
- debugging and performance monitoring values.

In Vx960, one important resource that is *not* part of a task's context is memory address space. In Vx960, all code executes in a single common address space. Giving each task its own memory space would require virtual-to-physical memory mapping, which runs counter to the Vx960 high-performance, real-time philosophy.

3.2.2 Task State Transition

The kernel maintains the current state of each task in the system. State transitions take place as the result of kernel function calls made by the application. When created, tasks enter the suspended state. Activation is necessary for a created task to enter the ready state. The activation phase is fast, enabling applications to pre-create tasks and activate them in a timely manner. A primitive is supplied for both creating and activating a task, referred to more as *spawning*. Tasks can be deleted from any state.

Vx960 kernel states are shown in the state transition diagram in Figure 3-1.

3.2.3 Task Scheduling

Multitasking requires a scheduling algorithm to allocate the CPU to ready tasks. Priority based preemptive scheduling is the default algorithm in Vx960, but applications can use round-robin selection as well. The following routines control task scheduling:

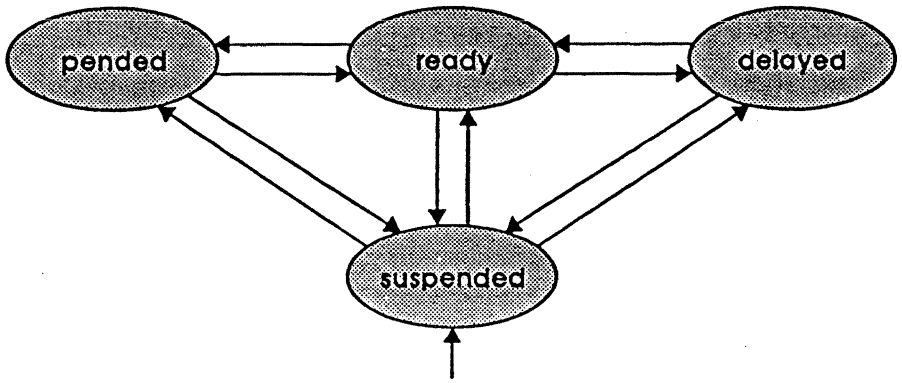
Table 3-1. Task Scheduler Control Routines

Call	Description
kernelTimeslice( )	Control round-robin scheduling.
taskPrioritySet( )	Change the priority of a task.
taskLock( )	Disable task rescheduling.
taskUnlock( )	Enable task rescheduling.

3.2.3.1 Preemptive Priority Scheduling

With a preemptive priority-based scheduler, each task is assigned a priority and the kernel ensures that the CPU is allocated to the highest priority task that is ready to run. The scheduling is *preemptive* in that if a task becomes ready to run that has

The highest priority task ready to run is executing.



ready	→	pended	*semTake( ) / msgQReceive( )*
ready	→	delayed	*taskDelay( )*
ready	→	suspended	*taskSuspend( )*
pended	→	ready	*semGive( ) / msgQSend( )*
pended	→	suspended	*taskSuspend( )*
delay	→	ready	expired delay
delay	→	suspended	*taskSuspend( )*
suspended	→	ready	*taskResume( ) / taskActivate( )*
suspended	→	pended	*taskResume( )*
suspended	→	delayed	*taskResume( )*

ready The state of a task that is not waiting for any resource other than the CPU.

pended The state of a task that is blocked due to the unavailability of some resource.

delayed The state of a task that is asleep for some duration.

suspended The suspended state is a secondary state used for debugging. Suspension does not inhibit state transition, only task execution. Thus *suspended-pended* tasks can still unblock and *suspended-delayed* tasks can still awaken. In either case the resulting state would be *suspended-ready.*

Figure 3-1. Task State Transitions

higher priority than the current task, the kernel saves the current task's context and switches to the context of the higher priority task.

Vx960 has 256 priority levels, numbered 0 through 255. Priority 0 is the highest and priority 255 is the lowest. Tasks are assigned a priority when created; however, while executing, a task can change its priority using *taskPrioritySet( )*. Dynamic prioritization of tasks enables an application to track precedence changes in the real world.

3.2.3.2 Round-Robin Scheduling

You can augment preemptive priority scheduling with round-robin scheduling. A round-robin scheduling algorithm attempts to share the CPU among all ready tasks of the *same priority*. Without round-robin scheduling, when multiple tasks of equal priority must share the processor, a single task can usurp the processor by never blocking, thus never giving other equal-priority tasks a chance to run.

Round-robin scheduling achieves fair allocation of the CPU to tasks of the same priority by an approach known as *time slicing*. The time line is cut into slices of equal duration and each slice is allocated in rotation to one of a group of tasks. The allocation is fair in that no task of a priority group is allocated a second slice of time before all tasks of a group have been given a slice. Round-robin scheduling can be enabled with the routine *kernelTimeSlice( )*, which takes a parameter for the "time slice," or interval, that each task is allowed to run before relinquishing the processor to another equal-priority task.

A run-time counter is kept for each task and incremented on every clock tick. When the specified time slice interval is completed, the counter is cleared and the task is placed at the tail of the queue of tasks at its priority. New tasks joining a given priority group are placed at the tail of the group with a run-time counter initialized to zero.

If a task is preempted by a higher priority task during its interval, its run-time count is saved and then restored when the task is again eligible for execution. Figure 3-2 shows round-robin scheduling for three tasks of the same priority: *t1*, *t2*, and *t3*. Task *t2* is preempted by a higher priority task *t4* but resumes at the count where it left off when *t4* is finished.

The *kernelTimeSlice( )* routine affects tasks at all priority levels. After time slicing is enabled, all tasks at the same priority level use round-robin scheduling.

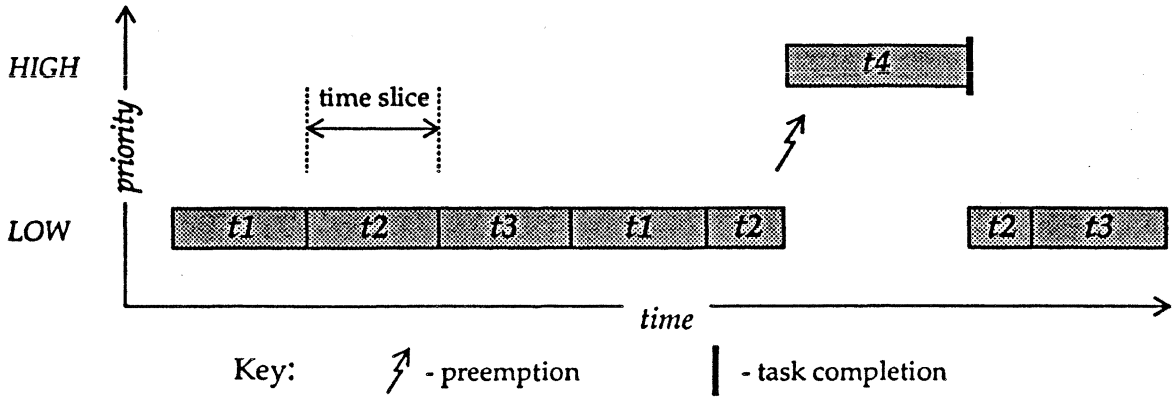


Figure 3-2. Round-Robin Scheduling

3.2.3.3 Preemption Locks

The Vx960 scheduler can be disabled and enabled on a per-task basis with the routines *taskLock( )* and *taskUnlock( )*. When a task disables the scheduler by calling *taskLock( )*, priority-based preemption does not occur while the task is running. However, if the task blocks or suspends, the scheduler selects the next highest priority eligible task to execute. When the preemption-locked task unblocks and begins running again, preemption again is disabled.

Preemption locks prevent task context switching but do not lock out interrupt handling. Preemption locks can be used to achieve mutual exclusion. See **3.3.2 Mutual Exclusion** for a more complete discussion.

3.2.4 Tasking Control

The following sections give an overview of Vx960's basic tasking routines, which are found in the Vx960 library **taskLib**. These routines provide the means for task creation, control, and information. See the manual entry for taskLib(1) for further discussion.

There are also many routines in **usrLib** that provide a more interactive interface to the tasking functions described here.

3.2.4.1 Task Creation and Activation

The following routines are used to create tasks:

Table 3-2. Task Creation Routines

Call	Description
taskSpawn( )	Spawn (create and activate) a new task.
taskInit( )	Initialize a new task.
taskActivate( )	Activate an initialized task.

The arguments to *taskSpawn*( ) are the new task's name (an ASCII string), priority, an "options" word, stack size, main routine address, and up to 10 arguments to be passed to the main routine as startup parameters:

```
id = taskSpawn ( name, priority, options, stacksize, main, arg1, ...arg10 );
```

If the specified routine takes fewer than 10 arguments, you should pass 0 to *taskSpawn*( ) for those unused arguments.

The *taskSpawn*( ) routine creates the new task context, which includes allocating the stack and setting up the task environment to call the main routine (a normal C routine) with the specified arguments. When the new task begins execution, it begins at the entry to the specified routine.

The *taskSpawn*( ) routine embodies the lower-level steps of allocation, initialization, and activation. The initialization and activation functions are provided by the routines *taskInit*( ) and *taskActivate*( ); however, these routines are only used when special circumstances dictate greater control over allocation or activation.

3.2.4.2 Task Names and IDs

When a task is spawned, the user specifies a task name, which is an ASCII string of arbitrary length. The system returns a task ID which is a four-byte handle to the task's data structures. Most Vx960 tasking routines take a task ID as the argument specifying a task. Vx960 uses a convention that a task ID of 0 always implies the calling task.

A task name should not conflict with any existing task name. Furthermore, to use the interactive shell fully, task names should not conflict with globally visible routine and variable names. Vx960 uses a convention of prefixing all task names with the letter "t" to avoid name conflicts.

You may not want to name some or any of your tasks. If a NULL pointer is supplied for the *name* argument of *taskSpawn*( ), then Vx960 assigns a name. The name is of the form "t*N*" where *N* is a decimal value that is incremented once for each unnamed task that is spawned.

When working interactively with the Vx960 shell, task names are resolved to their corresponding task ID to simplify interaction with existing tasks. See **9. Shell** for a more complete discussion of this feature.

The following **taskLib** routines manage task IDs and names:

Table 3-3. Task Name and ID Routines

Call	Description
taskName( )	Get the task name associated with a task ID.
taskNameToId( )	Look up the task ID associated with a task name.
taskIdSelf( )	Get the calling task's ID.
taskIdVerify( )	Verify the existence of a specified task.

3.2.4.3 Task Deletion and Deletion Safety

Tasks can be deleted from the system. Considerable care should be taken to avoid deleting tasks at inappropriate times. Tasks should be dormant before an application deletes them. Vx960 includes the following routines to delete tasks and protect tasks from unexpected deletion.

Table 3-4. Task Deletion Routines	
Call	**Description**
exit( )	Terminate and deallocate memory of calling task.
taskDelete( )	Terminate and deallocate memory of a task.
taskSafe( )	Make a task safe from deletion.
taskUnsafe( )	Make a task unsafe for deletion.

Tasks call *exit( )* if the entry routine specified during task creation returns. Alternatively, a task can call *exit( )* at any point to kill itself. A task can kill another task by calling *taskDelete( )*.

The routines *taskSafe( )* and *taskUnsafe( )* address problems that stem from unexpected deletion of tasks. The routine *taskSafe( )* protects a task from deletion by other tasks. This protection is often needed when a task executes in a critical region or engages a critical resource.

For example, a task could take a semaphore for exclusive access to some data structure. While executing inside the critical region, the task could be deleted by another task. Since the task was unable to complete the critical region, the data structure may have been left in a corrupt or inconsistent state. Furthermore, the semaphore can never be released by the task. Hence, the critical resource is now unavailable for use by any other task and is frozen.

Using *taskSafe( )* to protect the task that took the semaphore prevents such an outcome. Any task that tries to delete a task protected with *taskSafe( )* blocks. When finished with its critical resource, the protected task can make itself available for deletion by calling *taskUnsafe( )*, which unblocks any deleting task. To support nested deletion-safe regions, a count is kept of the number of times *taskSafe( )* and *taskUnsafe( )* are called. Deletion is allowed only when the count is zero, i.e., there are as many "unsafes" as "safes." Protection operates only on the calling task. A task cannot make another task safe or unsafe from deletion.

The following code fragment shows how a critical region of code can be protected using *taskSafe( )* and *taskUnsafe( )*:

```
taskSafe ();
semTake (semId, WAIT_FOREVER);
.
.    (critical region)
.
semGive (semId);
taskUnsafe ();
```

As shown in this example, deletion safety is often coupled with mutual exclusion. Thus, for convenience and efficiency, a special kind of semaphore, the mutual-exclusion semaphore, offers an option for deletion safety. See **3.3.2 Mutual Exclusion** for more information.

3.2.4.4 Task Options

When a task is spawned, the user specifies an option parameter that selects the options in the following table. The value of the option parameter is specified by performing a logical OR operation on the desired options.

VX_FP_TASK must be specified if the task performs any hardware floating-point operations. Similarly, VX_STDIO must be specified if the task uses any *stdio* functions other than *printf( )* and *sscanf( )*.

Table 3-5. Task Options

Name	Description
VX_UNBREAKABLE	Disable breakpoints for the task.
VX_FP_TASK	Execute with hardware floating-point support, if available.
VX_STDIO	Execute with *stdio* buffered I/O support.
VX_DEALLOC_STACK	Deallocate the stack on termination.

Task options can also be examined and altered after a task is spawned by means of the following routines. Currently only the VX_UNBREAKABLE option can be altered.

Table 3-6. Task Options Routines	
Call	**Description**
taskOptionsGet( )	Examine task options.
taskOptionsSet( )	Set task options.

3.2.4.5 Task Control

The following routines provide direct control over a task's execution:

Table 3-7. Task Control Routines	
Call	**Description**
taskSuspend( )	Suspend a task.
taskResume( )	Resume a task.
taskRestart( )	Restart a task.
taskDelay( )	Delay a task for a number of ticks.

Vx960's debugging facilities require task suspension and resumption routines. They are useful to freeze a task's state for examination. Delay operations provide a simple mechanism for a task to sleep for a fixed duration. Task delaying is often used for polling applications.

Tasks may require restarting during execution in response to some catastrophic error. The restart mechanism recreates a task with the original creation arguments. The Vx960 shell also uses this mechanism to restart the shell in response to a task-abort request.

3.2.4.6 Task Information

The following routines get information about a task by taking a snapshot of a task's context when called. The state of a task is dynamic, and the information may not be current unless the task is known to be dormant (i.e., suspended).

Table 3-8. Task Information Routines

Call	Description
taskIdListGet( )	Fill an array with the IDs of all active tasks.
taskInfoGet( )	Get information about a task.
taskPriorityGet( )	Examine the priority of a task.
taskRegsGet( )	Examine a task's registers.
taskRegsSet( )	Set a task's registers.
taskIsSuspended( )	Check if a task is suspended.
taskIsReady( )	Check if a task is ready to run.
taskTcb( )	Get a pointer to task's control block.

3.2.5 Task Error Status: *errno*

By convention, C library functions use a global mechanism for returning status codes when errors occur. A single global integer variable *errno* is set to an appropriate error number whenever a library function is to return information about an error. This convention is specified as part of the ANSI C standard.

Vx960 has a single pre-defined global variable *errno* that can be referenced directly by application code that is linked with Vx960 (either statically on the host or dynamically at load-time). However, to be useful in the multitasking environment of Vx960, each task must see its own version of *errno*. Thus *errno* is saved and restored by the kernel as part of each task's context every time a context switch occurs. Similarly, interrupt service routines must see their own versions of *errno*. Accomplish this by saving and restoring *errno* on the interrupt stack as part of the interrupt enter and

exit code provided automatically by Vx960 (see **3.4.1 Connecting Application Code to Interrupts**). Thus, regardless of Vx960 context, an error code can be stored or consulted with direct manipulation of the global variable *errno*.

Almost all Vx960 functions follow a convention that indicates simple success or failure of their operation by the actual return value of the function. Many functions return the status values OK (0) or ERROR (-1). Some functions which return a non-negative number (e.g., *open*( ) returns a file descriptor) also return ERROR to indicate an error. Functions which return a pointer return NULL (0) to indicate an error. In most cases, a function returning such an error indication also sets *errno* to the specific error code.

The global variable *errno* is never cleared by Vx960 routines. Thus, its value indicates the last error status set. When a Vx960 subroutine gets an error indication from a call to another routine, it returns its own error indication without modifying *errno*. Thus, the value of *errno* that was set in the lower-level routine remains available as the indication of error type.

For example, the Vx960 routine *intConnect*( ), which connects a user routine to a hardware interrupt, allocates memory by calling *malloc*( ) and builds the interrupt driver in this allocated memory. If *malloc*( ) fails because insufficient memory remains in the pool, it sets *errno* to a code indicating an insufficient memory error was encountered in the memory allocation library, memLib(1). The *malloc*( ) routine then returns NULL to indicate the failure. The *intConnect*( ) routine, receiving the NULL from *malloc*( ), then returns its own error indication of ERROR. However, it does not alter *errno*, leaving it at the "insufficient memory" code set by *malloc*( ).

Application developers are encouraged to employ this mechanism in their own subroutines, setting and examining *errno* as a debugging technique. See the manual entry errnoLib(1) for details on the composition of status values, and defined status values available to applications.

3.2.6 Task Exception Handling

Errors in program code or data can cause hardware exception conditions such as illegal instructions, bus or address errors, divide by zero, and so forth. The Vx960 exception handling package handles all such exceptions. The default exception handler suspends the task that caused the exception, and saves the state of task at the point of the exception. A description of the exception is displayed on standard output. The Vx960 kernel and other tasks continue uninterrupted. The suspended task can be examined with the usual Vx960 routines, including *ti( )* for task information and *tt( )* for a stack trace.

Tasks can also attach their own handlers for certain hardware exceptions through the UNIX-compatible *signal* facility. If a task has supplied a signal handler for an exception, the default exception handling described above is not performed. Signals are also used for signaling software exceptions as well as hardware exceptions. They are described in more detail in **3.3.7 Signals** below and in the manual entry for sigLib(1).

Note: The GNU/960 C runtime library includes signal routines (**signal** and **raise**) that are incompatible with the Vx960. Do not use these routines in your Vx960 applications.

3.2.7 Tasking Extensions

To allow additional task related facilities to be added to the system without modifying the kernel, Vx960 provides task create, switch, and delete *hooks* which allow additional routines to be invoked whenever a task is created, a task context switch occurs, or a task is deleted. There are spare fields in the task control block available for application extension of a task's context.

These hook routines are shown in the table below. For more information, see the manual entry for taskHookLib(1).

Table 3-9. Task Create, Switch, and Delete "Hooks"	
Call	**Description**
taskCreateHookAdd( )	Add routine to be called at every task create.
taskCreateHookDelete( )	Delete previously added task create routine.
taskSwitchHookAdd( )	Add routine to be called at every task switch.
taskSwitchHookDelete( )	Delete previously added task switch routine.
taskDeleteHookAdd( )	Add routine to be called at every task delete.
taskDeleteHookDelete( )	Delete previously added task delete routine.

3.2.8 Shared Code and Reentrancy

In Vx960, it is very common for a single copy of a subroutine or subroutine library to be invoked by many different tasks. For example, many tasks can call *printf( )*, but there is only a single copy of the subroutine in the system. This is called *shared code.* Shared code makes the system more efficient and easier to maintain.

However, shared code must be *reentrant.* A subroutine is reentrant if a single copy of the routine can be called from several task contexts simultaneously without conflict. Such conflict would occur when a subroutine modifies global or static variables, since there is only a single copy of the data and code. A routine's references to such variables can overlap and interfere in invocations from different task contexts.

All routines in Vx960 are made reentrant with the following techniques:

- use of dynamic stack variables
- guarding of global and static variables with semaphores
- use of task variables

These techniques should be used when writing application code that can be called from several task contexts simultaneously.

3.2.8.1 Dynamic Stack Variables

Many subroutines are *pure* code, having no data of their own except dynamic stack variables. They work exclusively on data provided by the caller as parameters. The linked-list library, lstLib, is a good example of this. Its routines operate on lists and nodes provided by the caller in each subroutine call.

Such subroutines of this kind are reentrant. Multiple tasks can use such routines without interfering with each other, since each task does indeed have its own stack.

3.2.8.2 Guarded Global and Static Variables

Some libraries encapsulate access to common data. An obvious example is the memory allocation library, memLib, which manages pools of memory to be used by many tasks. This library declares and uses its own static data variables to keep track of pool allocation.

This kind of library requires some caution since the routines are not reentrant. Multiple tasks invoking the routines in the library could interfere with access to common variables. Such libraries must be made reentrant by providing a *mutual-exclusion* mechanism to prohibit tasks from executing critical sections of code. The usual mutual-exclusion mechanism is the semaphore facility provided by semLib(1) and described in 3.3.3 Semaphores.

3.2.8.3 Task Variables

Some routines that can be called by multiple tasks may require global or static variables that should have a distinct value for each calling task. For example, tasks can reference a private buffer of memory and yet refer to it with the same global variable

To accommodate this, Vx960 provides a facility called *task variables*, which allows 4-byte variables to be added to a task's context, so that the value of such a variable is switched every time a task switch occurs to or from its owner task. Several tasks declare the same variable (4-byte memory location) as a task variable. Each of those tasks can then treat that single memory location as its own private variable. This facility is provided by the routines *taskVarAdd*( ), *taskVarDelete*( ), *taskVarSet*( ) and *taskVarGet*( ), which are described in the manual entry for taskVarLib(1).

Use this mechanism sparingly. Each task variable adds a few microseconds to the context switching time for its task since the value of the variable must be saved and restored as part of the task's context. It is prudent to collect all of a module's task variables into a single dynamically allocated structure, and then make all accesses to that structure be indirect through a single pointer. This pointer can then be the task variable for all tasks using that module.

3.2.8.4 Multiple Tasks with the Same Main Routine

With Vx960, it is possible to spawn several tasks with the same main routine. Each spawn creates a new task with its own stack and context. Each spawn can also pass the main routine different parameters to the new task. In this case, the same rules of reentrancy described in **3.2.8.3 Task Variables** apply to the entire task.

This is useful when the same function needs to be performed concurrently with different sets of parameters. For example, a routine that monitors a particular kind of equipment can be spawned several times to monitor several different pieces of that equipment. The arguments to the main routine could indicate which particular piece of equipment the task is to monitor.

3.2.9 Vx960 System Tasks

Vx960 includes several system tasks, described in the following sections.

3.2.9.1 The Root Task: *tUsrRoot*

The root task, *tUsrRoot*, is the first task executed by the kernel. In Vx960, the root task is in the module usrConfig and is used to initialize most Vx960 facilities. It spawns such tasks as the shell, the logging task, the exception task, the network task, and the *tRlogind* daemon. The root task terminates and is deleted when all initialization has been performed. You are free to add any necessary initialization of your own to the root task. For more information, see **8. Configuration.**

3.2.9.2 The Shell: *tShell*

The Vx960 shell, *tShell*, is also spawned as a task. The shell allows application developers to interact with Vx960 facilities and with their own application modules by invoking any subroutine that has been entered in the system symbol table. Routines that are called from the shell, rather than spawned, run in the context of the shell task.

3.2.9.3 The Logging Task: *tLogTask*

The log task, *tLogtask*, is used by Vx960 modules to log system messages without having to do I/O in the current task context. For more information, see **4.5.3 Message Logging** and the manual entry for logLib(1).

3.2.9.4 The Exception Task: *tExcTask*

The exception task, *tExcTask*, supports the Vx960 exception handling package by performing functions that cannot occur at interrupt level. It must have the highest priority in the system. Do not suspend, delete, or change the priority of this task. For more information, see the manual entry for excLib(1).

3.2.9.5 The Network Task: *tNetTask*

The *tNetTask* daemon handles the task-level functions required by the Vx960 network.

3.2.9.6 The Remote Login Daemon: *tRlogind*

The remote login daemon, *tRlogind*, allows remote users to log in to Vx960. It accepts remote login requests from other Vx960 or host systems, and causes the shell's input and output to be redirected to the remote user. A *tty*-like interface is provided to the remote user through the use of the Vx960 pseudo-terminal driver, ptyDrv. For more information, see **4.6.2 Serial I/O Devices (Terminal and Pseudo-Terminal Devices)** and the manual entry for ptyDrv(2).

3.2.9.7 The Telnet Daemon: *tTelnetd*

The telnet daemon, *tTelnetd*, allows remote users to log into Vx960 via **telnet**. It accepts remote connection requests from any other system, and causes the shell's input and output to be redirected to the remote user. A *tty*-like interface is provided to the remote user through the use of the Vx960 pseudo-terminal driver, **ptyDrv**. See **4.6.2 Serial I/O Devices (Terminal and Pseudo-Terminal Devices)** and the manual entry for **ptyDrv** for further explanation.

3.2.9.8 The Portmap Daemon: *tPortmapd*

The *tPortmapd* daemon is an RPC server that acts as a central registrar for RPC servers running on the same machine. RPC clients query the *tPortmapd* daemon to find out how to contact the various servers.

3.2.9.9 The Remote Debugging Server: *tRdbTask*

The *tRdbTask* services RPC requests made by remote source-level debuggers.

3.3 Intertask Communications

The intertask communication facilities complement the multitasking functions described previously. These facilities permit independent tasks to coordinate their actions.

Vx960 supplies a rich set of intertask communication mechanisms, including:

- *shared memory*, for simple sharing of data
- *semaphores*, for basic mutual exclusion and synchronization
- *message queues* and *pipes*, for intertask message passing within a CPU
- *sockets* and *remote procedure calls*, for network-transparent intertask communication
- *signals*, for exception handling.

3.3.1 Shared Memory

The most obvious way for tasks to communicate in Vx960 is by accessing shared data structures. Because all tasks in Vx960 exist in a single linear address space, sharing data structures between tasks is trivial. Global variables, linear buffers, ring buffers, linked lists, and pointers can be referenced by code running in different contexts.

3.3.2 Mutual Exclusion

While a shared address space simplifies exchange of data, interlocking access to memory is crucial to avoid contention. Many mechanisms exist for obtaining exclusive access to a resource and differ only in the scope for which the exclusion applies. Methods of mutual exclusion include disabling interrupts, disabling preemption, and resource locking with semaphores.

3.3.2.1 Interrupt Locks and Latency

The most powerful method available for mutual exclusion is the disabling of interrupts. Such a lock guarantees exclusive access to the CPU.

```
funcA ()
    {
    int lock = intLock();
    .
    .    (critical region that cannot be interrupted)
    .
    intUnlock (lock);
    }
```

While this solves problems involving mutual exclusion with interrupt service routines, it is inappropriate as a general-purpose mutual exclusion method for most real-time systems because of the inherent inability to respond to external events for the duration of these locks. High interrupt latency is unacceptable in applications for which deterministic response is a requirement.

However, interrupt locking can sometimes be necessary where mutual exclusion involves interrupt service routines. In any situation, the duration of interrupt lockouts should be kept short.

3.3.2.2 Preemptive Locks and Latency

Disabling preemption offers a somewhat less restrictive form of mutual exclusion. While no other task is allowed to preempt the current executing task, interrupt service routines are able to execute.

```
funcA ()
    {
    taskLock ();
    .
    .    (critical region that cannot be preempted)
    .
    taskUnlock ();
    }
```

However, this method can also lead to unacceptable real-time response. Tasks of higher priority are unable to execute until the locking task leaves the critical region, even though the higher priority task was not itself involved with the critical region. While this kind of mutual exclusion is simple, the duration should be kept short. A better mechanism is provided by semaphores, discussed in the following sections.

3.3.3 Semaphores

Vx960 semaphores are optimized and provide the fastest intertask communication mechanism in Vx960. They are the primary means for addressing the requirements of both mutual exclusion and task synchronization:

- When used for *mutual exclusion*, Vx960 semaphores interlock access to shared resources. They provide mutual exclusion with finer granularity than either interrupt disabling or preemptive locks, discussed above.
- When used for *synchronization*, Vx960 semaphores coordinate a task's execution with an external event.

There are three semaphore types provided in Vx960, each optimized to address a different class of problem:

- The basic *binary* semaphore is the fastest, most general-purpose semaphore.
- The extended *mutual-exclusion* semaphore is a specialized form of the binary semaphore that addresses problems inherent in mutual exclusion.
- The *counting* semaphore works like the binary semaphore except that it keeps track of the number of times a semaphore has been given.

Table 3-10. Semaphore Types	
Type	**Uses**
binary	General; for synchronization or mutual exclusion.
mutual exclusion	For priority inheritance, deletion safety, and recursion.
counting	For guarding multiple instances of a resource.

3.3.3.1 Semaphore Control

Instead of defining a complete set of semaphore control functions for each type of semaphore, Vx960 provides a single uniform interface for semaphore control. An entire set of semaphore calls can be changed from one type of semaphore to another. Only the creation routines are specific to the semaphore type used. The following semaphore control functions are provided:

Table 3-11. Semaphore Control Routines	
Call	**Description**
semBCreate( )	Allocate and initialize a binary semaphore.
semMCreate( )	Allocate and initialize a mutual-exclusion semaphore.
semCCreate( )	Allocate and initialize a counting semaphore.
semDelete( )	Terminate and deallocate a semaphore.
semTake( )	Take a semaphore.
semGive( )	Give a semaphore.
semFlush( )	Flush all pended tasks off a semaphore.

The *sem*[*BMC*]*Create*( ) routines return a semaphore ID that serves as a handle on the semaphore during subsequent use by the other semaphore-control routines.

3.3.3.2 Binary Semaphores

The general-purpose binary semaphore is capable of addressing the requirements of both forms of task coordination: mutual exclusion and synchronization. The binary semaphore has the least overhead associated with it, making it applicable to high performance requirements. The mutual-exclusion semaphore described below in **3.3.3.3 Mutual-Exclusion Semaphores** is also a binary semaphore, but it has been tailored to address problems inherent to mutual exclusion. Alternatively, the binary semaphore can be used for mutual exclusion if the advanced features of the mutual-exclusion semaphore are deemed unnecessary.

A binary semaphore can be viewed as a cell in memory that can be full or empty. When a task takes a binary semaphore, using *semTake*( ), the outcome depends on whether the semaphore was full or empty at the time of the call. If the semaphore was full, then the semaphore is made empty and the task continues execution. If the semaphore was empty, then the task is put on a queue of blocked tasks and enters a state of pending on the availability of the semaphore.

When a task gives a binary semaphore, using *semGive*( ), the outcome also depends on whether the semaphore was full or empty at the time of the call. If the semaphore was already full, giving the semaphore has no effect at all. If the semaphore was empty and no task was waiting to take it, then the semaphore is made full. If the semaphore was empty and one or more tasks were pending on its availability, then the first task in the queue of blocked tasks is unblocked, and the semaphore is left empty.

- Mutual Exclusion

Binary semaphores interlock access to a shared resource. Unlike disabling interrupts or preemptive locks, binary semaphores limit the scope of the mutual exclusion to the associated resource. In this technique, a semaphore is created to guard the resource. Initially the semaphore is full.

```
SEM_ID semMutex;

/* create a binary semaphore that is initially full */
semMutex = semBCreate (SEM_Q_PRIORITY, SEM_FULL);
```

When a task wants to access the resource, it must first take that semaphore. So long as the task keeps the semaphore, all other tasks seeking access to the resource are blocked from execution. When the task is finished with the resource, it gives back the semaphore allowing another task to use the resource.

Thus all accesses to a resource requiring mutual exclusion are bracketed with *semTake*( ) and *semGive*( ) pairs:

```
semTake (semMutex, WAIT_FOREVER);
.
.   (critical region, only accessible by a single task at a time)
.
semGive (semMutex);
```

- **Synchronization**

When used for task synchronization, a semaphore can represent a condition or event that a task is waiting for. Initially the semaphore is empty. A task or interrupt service routine signals the occurrence of the event by giving the semaphore. Another task waits for the semaphore by calling *semTake*( ). The waiting task is blocked until the event occurs and the semaphore is given.

Note the difference in sequence when semaphores are used for mutual exclusion and when they are used for synchronization. For mutual exclusion, the semaphore is initially full, and each task first takes and then gives back the semaphore. For synchronization, the semaphore is initially empty, and one task waits to take the semaphore, which is given by another task.

The following is an example of using semaphores for task synchronization:

```
SEM_ID syncSem; /* ID of sync semaphore */

init ()
    {
    intConnect (..., eventInterruptSvcRout, ...);
    syncSem = semBCreate (SEM_Q_FIFO, SEM_EMPTY);
    taskSpawn (..., task1);
    }

task1 ()
    {
    ...
    semTake (syncSem, WAIT_FOREVER); /* wait for event to occur */
    .../* process event */
    }

eventInterruptSvcRout ()
    {
    ...
    semGive (syncSem); /* let task 1 process event */
    ...
    }
```

In the above example, when the *init( )* routine is called, the binary semaphore is created, an interrupt service routine is attached to an event, and a task is spawned to process the event. The routine *task1( )* runs until it calls *semTake( )*. It remains blocked at that point until an event causes the interrupt service routine to call *semGive( )*. When the interrupt service routine completes, *task1( )* executes to process the event. There is an advantage to handling event processing within the context of a dedicated task. Less processing takes place at interrupt level, thereby reducing interrupt latency. This model of event processing is recommended for real-time applications.

3.3.3.3 Mutual-Exclusion Semaphores

The mutual-exclusion semaphore is a specialized binary semaphore designed to address issues inherent in mutual exclusion, including priority inversion, deletion safety, and recursive access to resources.

The fundamental behavior of the mutual-exclusion semaphore is identical to the binary semaphore, with the following exceptions:

- It can only be used for mutual exclusion.
- It can only be given by the task that took it.
- It can not be given from interrupt-level code.
- The *semFlush( )* operation is illegal.

- **Priority Inversion**

Priority inversion arises when a higher-priority task is forced to wait an indefinite period of time for the completion of a lower-priority task. Consider the following scenario: *t1*, *t2*, and *t3* are tasks of high, medium, and low priority, respectively. t3 has acquired some resource by taking its associated binary guard semaphore. When *t1* preempts *t3* and contends for the resource by taking the same semaphore, it becomes blocked. If we could be assured that *t1* would be blocked no longer than the time it takes *t3* to finish with the resource, there would be no problem since the resource cannot be preempted. However, the low-priority task is vulnerable to preemption by medium-priority tasks (like *t2*), which could inhibit *t3* from relinquishing the resource. This condition could persist, blocking *t1* for an indefinite period of time. See the diagram in Figure 3-3.

The mutual-exclusion semaphore has an additional option SEM_INVERSION_SAFE, which enables a *priority inheritance* algorithm. The priority inheritance protocol assures that a task which owns a resource executes at the priority of the highest pri-

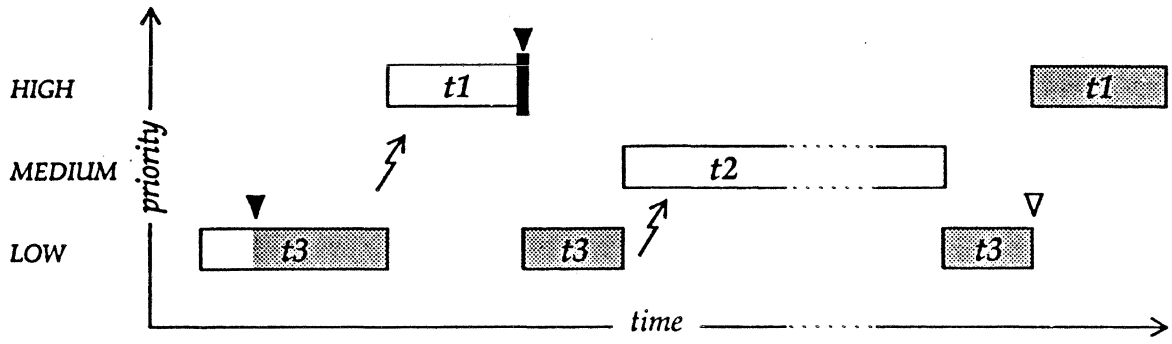


Figure 3-3. Priority Inversion

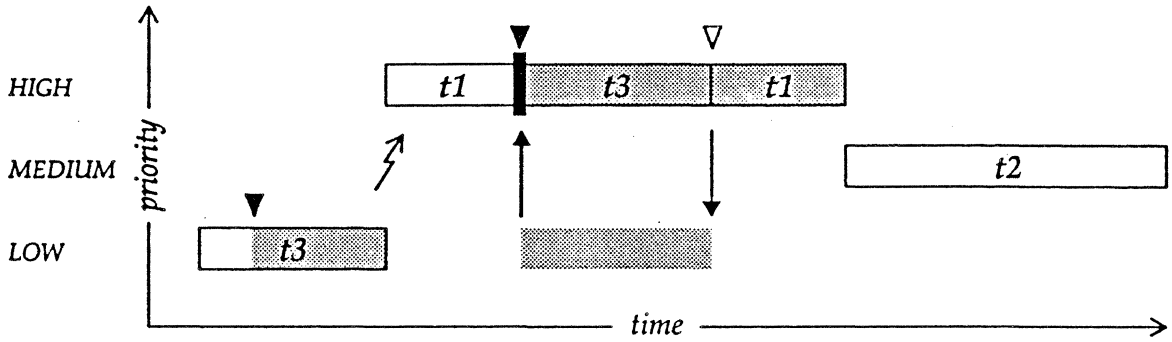


Figure 3-4. Priority Inheritance

Key:
▼ - take semaphore
∇ - give semaphore
▒ - own semaphore
↯ - preemption
↑↓ - priority inheritance/release
▮ - block

ority task blocked on that resource. When execution is complete, the task gives up the resource and returns to its normal or standard priority. Hence, the inheriting task is protected from preemption by any intermediate-priority tasks. Priority inheritance solves the problem of priority inversion by elevating the priority of *t3* to the priority of *t1* during the time *t1* is blocked on *t3*. This protects *t3*, and indirectly *t1*, from preemption by *t2*. See the diagram in Figure 3-4.

- **Deletion Safety**

  Another problem of mutual exclusion involves task deletion. Within a critical region guarded by semaphores, it is often desirable to protect the executing task from unexpected deletion. Deleting a task executing in a critical region can be catastrophic. The resource could be left in a corrupted state and the semaphore guarding the resource left unavailable, preventing all access to the resource.

  The primitives *taskSafe*( ) and *taskUnsafe*( ) provide one solution to task deletion. However, the mutual-exclusion semaphore offers the option SEM_DELETE_SAFE, which enables an implicit *taskSafe*( ) with each *semTake*( ), and a *taskUnsafe*( ) with each *semGive*( ). In this way, a task can be protected from deletion while it has the semaphore. This option is more efficient than the primitives *taskSafe*( ) and *taskUnsafe*( ), as the resulting code requires fewer entrances to the kernel.

- **Recursive Resource Access**

  Mutual-exclusion semaphores can be taken recursively. This means that the semaphore can be taken more than once by the task that owns it before being released. Recursion is useful for a set of routines that need mutually exclusive access to a resource, but may need to call each other. This is possible because the system keeps track of which task currently owns the mutual-exclusion semaphore.

  Before being released, a mutual-exclusion semaphore taken recursively must be given the same number of times it has been taken. This is tracked by a count which is incremented with each *semTake*( ) and decremented with each *semGive*( ).

The example below illustrates recursive use of a mutual exclusion semaphore. Function A requires access to a resource which it acquires by taking *mySem*; function A may also need to call function B, which also requires *mySem*:

```
SEM_ID mySem;

mySem = semMCreate (...);

funcA ()
    {
    semTake (mySem, WAIT_FOREVER);
    ...

    funcB ();
    ...
    semGive (mySem);
    }

funcB ()
    {
    semTake (mySem, WAIT_FOREVER);
    ...
    semGive (mySem);
    }
```

3.3.3.4 Counting Semaphores

Counting semaphores are another means to implement task synchronization and mutual exclusion. The counting semaphore works like the binary semaphore except that it keeps track of the number of times a semaphore has been given. Every time a semaphore is given, the count is incremented; every time a semaphore is taken, the count is decremented. When the count reaches zero, a task that tries to take the semaphore is blocked. As with the binary semaphore, if a semaphore is given and a task is blocked, it becomes unblocked. However, if a semaphore is given and no tasks are blocked, then the count is incremented. This means that a semaphore that has been given twice can be taken twice without blocking, in contrast to a binary semaphore.

Counting semaphores are useful for guarding resources of which there are multiple copies. For example, the use of five tape drives could be coordinated using a counting semaphore with an initial count of five, or a ring buffer with 256 entries could be implemented using a counting semaphore with an initial count of 256. The initial count is specified as an argument to the *semCCreate*( ) routine.

3.3.3.5 Special Semaphore Options

- **Timeouts**

  All semaphore types include the ability to time out from the pended state. This is controlled by a parameter to *semTake*( ) that specifies the amount of time in ticks that the task is to wait in a blocked state. If the task takes the semaphore within the allotted time, *semTake*( ) returns OK; if it times out before taking the semaphore, it returns ERROR. A timeout value of WAIT_FOREVER means wait indefinitely; a value of NO_WAIT means do not wait at all.

- **Queues**

  All semaphore types include the ability to select the queuing mechanism employed for tasks blocked on a semaphore. They can be queued based on either of two criteria: first-in-first-out (FIFO) order, or priority order. Priority ordering better preserves the intended priority structure of the system at the expense of some overhead in *semTake*( ) in sorting the tasks by priority. A FIFO queue requires no priority sorting overhead and leads to constant-time performance. The selection of queue type is specified during semaphore creation with *sem*[*BMC*]*Create*( ). Semaphores using priority inheritance (SEM_INVERSION_SAFE option) must select priority-order queuing.

3.3.4 Message Queues

Modern real-time applications are constructed as a set of independent but cooperating tasks. While semaphores provide a high-speed mechanism for the synchronization and interlocking of tasks, often a higher-level mechanism is necessary to allow cooperating tasks to communicate with each other. In Vx960, the primary intertask communication mechanism within a single CPU is *message queues*. Message queues allow a variable number of messages, each of variable length, to be queued in first-in-first-out order. Any task or interrupt service routine can send messages to a message queue. Any task can receive messages from a message queue. Multiple tasks can send to and receive from the same message queue. Full-duplex communication between two tasks requires two message queues, one for each direction.

3.3.4.1 Creating and Using Message Queues

Message queues are created and deleted with the routines shown in the table below.

A message queue is created with *msgQCreate( )*. Its parameters specify the maximum number of messages that can be queued to the message queue and the maximum length in bytes of each message. Enough buffer space is preallocated for the specified number and length of messages.

Table 3-12. Message Queue Creation/Deletion/Control

Call	Description
msgQCreate( )	Allocate and initialize a message queue.
msgQDelete( )	Terminate and deallocate a message queue.
msgQSend( )	Send a message to a message queue.
msgQReceive( )	Receive a message from a message queue.

A task or interrupt service routine sends a message to a message queue with *msgQSend( )*. If no tasks are waiting for messages on that queue, the message is added to the queue's buffer of messages. If any tasks are already waiting for a message from that message queue, the message is delivered to the first waiting task.

A task receives a message from a message queue with *msgQReceive( )*. If messages are already available in the message queue's buffer, the first message is dequeued and returned to the caller. If no messages are available, then the calling task blocks and be added to a queue of tasks waiting for messages. This queue of waiting tasks can be ordered either by task priority or FIFO, as specified in an option parameter when the queue is created.

- **Timeouts**

Both *msgQSend( )* and *msgQReceive( )* take timeout parameters. When sending a message, if no buffer space is available to queue the message, the timeout specifies how many ticks to wait for space to become available. When receiving a message, if no message is available, the timeout specifies how many ticks to wait for a message to become available. As with semaphores, the value of the timeout parameter can

have the special values of NO_WAIT (0) meaning to always return immediately or WAIT_FOREVER (-1) meaning to never time out the function.

- **Urgent Messages**

  The *msgQSend( )* function allows specification of the priority of the message as either normal (MSG_PRI_NORMAL) or urgent (MSG_PRI_URGENT). Normal priority messages are added to the tail of the list of queued messages, while urgent priority messages are added to the head of the list.

3.3.4.2 Servers and Clients with Message Queues

Real-time systems are often structured using a *client-server* model of tasks. In this model, server tasks accept requests from client tasks to perform some service and return some reply. The requests and replies are made in the form of intertask messages. In Vx960, message queues, or pipes (see below), are a natural way to implement this.

For example, client-server communications could be implemented as follows. Each server task would create a message queue to receive request messages from clients. Each client task would create a message queue to receive reply messages from servers. Each request message would include a field containing the *msqQId* of the client's reply message queue. A server task's "main loop" would consist of reading request messages from its request message queue, performing the request, and sending a reply to the client's reply message queue.

The same architecture could be achieved with pipes instead of message queues. Of course, there can be many variations of this example, tailored to the needs of the particular application.

3.3.5 Pipes

Pipes provide an alternative interface to the message queue facility that goes through the Vx960 I/O system. Pipes are virtual I/O devices managed by the **pipeDrv** driver. The routine *pipeDevCreate*( ) creates a pipe device and the underlying message queue associated with that pipe. The call specifies the name of the created pipe, the maximum number of messages that can be queued to the pipe, and the maximum length of each message:

```
status = pipeDevCreate ("/pipe/name", max_msgs, max_length);
```

The created pipe is a named I/O device. Tasks can use the standard I/O routines to open, read, and write pipes, and invoke *ioctl* functions. As they do with other I/O devices, tasks block when they read from an empty pipe until data is available, and block when they write to a full pipe until there is space available. Like message queues, interrupt service routines can write to a pipe, but cannot read from a pipe.

As I/O devices, one important feature that pipes provide that message queues cannot, is the ability to be used with *select*( ). This routine allows a task to wait for data to be available on any set of I/O devices. The *select*( ) routine also works with other asynchronous I/O devices including network sockets and serial devices. Thus, using *select*( ), a task can wait for data on a combination of several pipes, sockets, and serial devices.

Pipes, like message queues, are a natural way to implement a client-server model of intertask communications. See the section above, **3.3.4.2 Servers and Clients with Message Queues.**

3.3.6 Network Intertask Communication

3.3.6.1 Sockets

In Vx960, the basis of intertask communications across the network is *sockets*. A socket is an endpoint for communications between tasks; data is sent from one socket to another. When you create a socket, you specify an Internet communications protocol that is used to transmit the data. Vx960 supports the Internet protocols TCP and UDP. Vx960 socket facilities are source compatible with BSD 4.3 UNIX.

TCP provides reliable, guaranteed, two-way transmission of data via *stream sockets*. In a stream-socket communication, two sockets are connected, allowing a reliable

byte-stream to flow between them in each direction like a circuit. For this reason TCP is often referred to as a virtual circuit protocol.

UDP provides a simpler but less robust form of communication. In a UDP communication, data is sent between sockets in separate, unconnected, individually addressed packets called *datagrams*. A process creates a datagram socket and binds it to a particular port number. There is no notion of a UDP connection. Any UDP socket, on any host in the network, can send messages to any other UDP socket by specifying its Internet address and port number.

One of the biggest advantages of socket communications is that it is homogeneous. Socket communications among processes are the same regardless of the location of the processes in the network, or the operating system under which they are running. Processes can communicate within a single CPU, across a backplane, across an Ethernet, or across any connected combination of networks. Socket communications can occur between Vx960 tasks and UNIX processes in any combination. In all cases, the communications look identical to the application, except, of course, for the speed of the communications.

For more information, see **6.2.5 Sockets** and the manual entry for **sockLib(1)**.

3.3.6.2 Remote Procedure Calls (RPC)

Remote Procedure Calls (RPC) is a facility which allows a process on one machine to call a procedure which is executed by another process on either the same machine or a remote machine. Internally, RPC uses sockets as the underlying communication mechanism. Thus with RPC, Vx960 tasks and UNIX processes can invoke routines that are executed on other Vx960 or UNIX machines, in any combination.

As discussed in the sections above on message queues and pipes, many real-time systems are structured with a client-server model of tasks. In this model, client tasks request services of server tasks, and then wait for their reply. RPC formalizes this model and provides a standard protocol for passing requests and returning replies. Also, RPC includes tools to help generate the client interface routines and the server skeleton.

For more information on RPC, see **6.2.6 Remote Procedure Calls**.

3.3.7 Signals

Vx960 supports UNIX BSD-style *signals*. Signals asynchronously alter the control flow of a task. Any task or interrupt service routine can raise a signal for a particular task. The task being signaled suspends its current thread of execution and begins execution of a task-specified signal handler routine, the next time the task is scheduled to run. Signals are more appropriate for error and exception handling than as a general-purpose intertask communication mechanism.

The primary signal routines are shown in the table below. These routines are source compatible with UNIX BSD 4.3 signal functions, hence the odd name of *kill( )* for the "raise signal" function. For more information on these and other signal functions, see the manual entry sigLib(1).

Table 3-13. Signal Functions

Call	Description
sigvec( )	Set signal handler routine for a signal.
kill( )	Raise a signal.
sigsetmask( )	Set mask of blocked signals.
sigblock( )	Block a single signal.

In many ways, signals are analogous to hardware interrupts. The signal facility provides a set of 31 distinct signals. A *signal handler* binds to a particular signal with *sigvec( )*, in much the same way that an interrupt service routine is connected to an interrupt vector with *intConnect( )*. A signal can be asserted by calling *kill( )*. This is analogous to the occurrence of an interrupt. The routines *sigsetmask( )* and *sigblock( )* let signals be selectively inhibited.

Certain signals are associated with hardware exceptions. For example, bus errors, illegal instructions, and floating-point exceptions raise specific signals.

3.4 Interrupt Service Code

Hardware interrupt handling is of key significance in real-time systems since it is through interrupts that the system is informed of external events. For the fastest possible response to interrupts, interrupt service code in Vx960 runs in a special context outside of the context of other tasks. Thus, handling an interrupt does not involve a task context switch.

3.4.1 Connecting Application Code to Interrupts

System hardware interrupts other than those used by Vx960 are available for use by application developers. Vx960 supplies a routine, *intConnect( )*, that allows C functions to be connected to any interrupt. The arguments to *intConnect( )* are the byte offset of the interrupt vector to connect to, the address of the C function to be connected, and an argument to be passed to the C function. When an interrupt occurs whose vector has been established in this way, the connected C function is called at interrupt level with the specified argument. To return from interrupt, the connected function returns. A routine connected to an interrupt in this way is referred to as *interrupt service routine* (ISR) or *interrupt handler.*

Interrupts cannot vector to C functions. Instead, the *intConnect( )* routine builds a small amount of code which saves the necessary registers, sets up the stack with the argument to be passed, and calls the connected function. Upon return from the function, it restores the registers and stack, and exits the interrupt.

The interrupt routines, shown below, are provided in intLib(1).

Table 3-14. Interrupt Routines	
Call	**Description**
intConnect( )	Connect C routine to interrupt vector.
intContext( )	Return TRUE if called from interrupt level.
intCount( )	Get current interrupt nesting depth.
intLevelSet( )	Set processor interrupt mask level.
intLock( )	Disable interrupts.
intUnlock( )	Enable intLock.
intVecBaseGet( )	Get the vector base address.
intVecSet( )	Set an exception vector.
intVecGet( )	Get an exception vector.

3.4.2 Interrupt Stack

All ISR code uses the same *interrupt stack*. This stack is allocated and initialized by the system at startup according to specified configuration parameters. It must be large enough to handle the worst possible combination of nested interrupts. The facility *checkStack( )* shows interrupt stack utilization. See **8. Configuration** for details of setting the interrupt stack size.

3.4.3 Special Limitations of Interrupt Code

Many Vx960 facilities are available to interrupt service code, but there are some important limitations. These limitations stem from the fact that interrupt service code does not run in a regular task context: it has no task control block, for example, and all ISRs share a single stack.

For this reason, the basic restriction on ISRs is that they must not invoke functions that might cause "blocking" of the caller. For example, they must not try to take a

semaphore, because if the semaphore is unavailable, the kernel trys to "block" the caller. However, ISRs may give semaphores that tasks are waiting on.

The memory facilities *malloc( )* and *free( )* take a semaphore and are therefore not interrupt callable, nor are routines which make calls to *malloc( )* and *free( )*. For example all creation and deletion routines are restricted from interrupt-level use.

An ISR also must not perform I/O through Vx960 drivers. Although there are no inherent restrictions in the Vx960 I/O system, most device drivers require a task context since they might block the caller to wait for the device. An important exception is the Vx960 pipe driver which has been designed to permit writes by interrupt service code.

Vx960 supplies a logging facility that allows text messages to be logged to the system console via a logging task. This mechanism has been designed to be callable by ISRs. This is the most common way to print out messages from interrupt service code. For more information, see the manual entry for logLib(1).

On i960 microprocessors with on-chip floating-point support (e.g., 80960 KB and 80960 SB), an ISR must not execute code that makes use of this support. In Vx960, the interrupt driver code created by *intConnect( )* does not save and restore floating-point registers; thus, ISRs must not include floating-point instructions. If an ISR requires floating-point instructions, the floating-point registers must be saved and restored using routines in fppALib(1).

All Vx960 utility libraries, such as the linked-list and ring buffer libraries, are usable in interrupt service code. As was mentioned in the section on *errno* above (**3.2.5 Task Error Status: errno**), the global variable *errno* is saved and restored as a part of the interrupt enter and exit code generated by the Vx960 *intConnect( )* facility. Thus *errno* can be referenced and modified by ISRs just as in any other code.

3.4.4 Exceptions At Interrupt Level

When a task causes a hardware exception such as illegal instruction or bus error, the task is suspended and the rest of the system continues uninterrupted. However, when an ISR causes such an exception, there is no safe recourse for the system to handle the exception. The ISR has no context that can be suspended. Instead, Vx960 stores the description of the exception in a special location in low memory and executes a system restart.

The Vx960 boot ROMs test for the presence of the exception description in low memory and if detected, print it out on the system console. The **e** command in the boot ROMs redisplays the exception description.

3.4.5 Interrupt-to-Task Communication

While it is important that Vx960 support direct connection of ISRs that run at interrupt level, interrupt events propagate to task-level code. Many Vx960 facilities are not available to interrupt-level code, including I/O to any device other than pipes. The following techniques can be used to communicate from interrupt service code to task-level code.

Shared Memory and Ring Buffers
Interrupt service code can share variables, buffers, and ring buffers with task-level code.

Semaphores Interrupt service code can give semaphores that tasks take and wait for.

Message Queues Interrupt service code can send messages to message queues for tasks to receive. If the queue is full, the message is discarded.

Pipes Interrupt service code can write messages to pipes that tasks read. Tasks and interrupt service code can write to the same pipes. However, if the pipe is full, the message written is discarded since the interrupt service code cannot block. Interrupt service code must not invoke any I/O function on pipes other than *write*( ).

Signals Interrupt service code can signal tasks, causing asynchronous scheduling of their signal handlers.

Note: Table 3-15 lists the routines callable by the interrupt service routines. The libc routines follow the arrangement set up in *C: A Reference Manual,* Samuel P. Harbison and Guy L. Steele Jr., Second Edition. Prentice Hall, Inc., Englewood Cliffs, New Jersey. 1987.

Table 3-15. Routines Callable by Interrupt Service Routines

Library	Routines
bLib	All routines.
errnoLib	*errnoGet( ), errnoSet( )*
fppALib.s	*fppSave( ), fppRestore( )*
intLib	*intContext( ), intCount( ), intVecSet( ), intVecGet( )*
libc	Math Functions: *abs( ), fabs( ), labs( )* *div( ), ldiv( )* *ceil( ), floor( ), fmod( )* *exp( ), log( ), log10( )* *frexp( ), ldexp( ), modf( )* *pow( ), sqrt( )* *srand( )* *cos( ), sin( ), tan( )* *acos( ), asin( ), atan( ), atan2( )* *cosh( ), sinh( ), tanh( )* *atanh( ), poly( ), mod( ), rint( ), sign( ), matherr( ), max( ), min( )* *dabs( ), acosh( )* Search and Sort Functions: *bsearch( )* String Functions: *strcat( ), strncat( )* *strcmp( ), strncmp( )* *strcpy( ), strncpy( )* *strlen( )* *strchr( ), strpos( ), strrchr( ), strrpos( )* *strspn( ), strcspn( ), strpbrk( ), strrpbrk( )* *strstr( )* *strtod( ), strtol( ), strtoul( )* *atof( ), atoi( ), atol( )* *strcoll( ), strxfrm( )*

Table 3-15. Routines Callable by Interrupt Services Routines (continued)

Library	Routines
	Memory Functions: *memchr( )* *memcmp( )* *memcpy( ), memccpy( ), memmove( ), bcopy( )* *memset( ), bzero( )* Character Functions: *isalnum( ), isalpha( ), isascii( ), iscntrl( )* *isdigit( ), isxdigit( )* *isprint( )* *islower( ), isupper( )* *isspace( )* *toascii( )* *tolower( ), toupper( )*
logLib	*logMsg( )*
lstLib	All routines.
msgQLib	*msgQSend( )*
pipeDrv	*write( )*
rngLib	All routines except *rngCreate( )*.
semLib	*semGive( ), semFlush( )*
sigLib	*kill( ), sigRaise( )*
strLib	All routines.
taskLib	*taskSuspend( ), taskResume( ), taskPrioritySet( ),* *taskPriorityGet( ), taskIdVerify( ), taskIdDefault( ),* *taskIsReady( ), taskIsSuspended( ), taskTcb( )*
tickLib	*tickAnnounce( ), tickSet( ), tickGet( )*
vxLib	*vxTas( )*
wdLib	*wdStart( ), wdCancel( )*

3.5 Watchdog Timers

Vx960 includes a watchdog timer mechanism that allows arbitrary C functions to be connected to a specified time delay. Watchdog timers are maintained as part of the system clock interrupt service routine. Functions invoked by watchdog timers execute as interrupt service code at the interrupt level of the system clock. The restrictions on interrupt service code apply to routines connected to watchdog timers.

The watchdog timer functions in the following table are provided by the **wdLib** library.

Table 3-16. Watchdog Timer Functions

Call	Description
wdCreate( )	Allocate and initialize a watchdog timer.
wdDelete( )	Terminate and deallocate a watchdog timer.
wdStart( )	Start a watchdog timer.
wdCancel( )	Cancel a currently counting watchdog timer.

A watchdog timer is first created by calling the Vx960 routine *wdCreate*( ). Then the timer can be started by calling *wdStart*( ), which takes as arguments the number of ticks to delay, the C function to call, and an argument to be passed to that function. After the specified number of ticks have elapsed, the C function is called with the specified argument. You can cancel Watchdog timers any time before the specified delay has elapsed by calling *wdCancel*( ).

4

I/O System

Contents

4

I/O System

4.1 Introduction

The Vx960 I/O system is designed to present a simple, uniform, device-independent interface to any kind of device including:

- character-oriented devices such as terminals or communications lines
- random-access block devices such as disks
- virtual devices such as intertask *pipes* and *sockets*
- monitor and control devices such as digital/analog I/O devices
- network devices that give access to remote devices on other computers.

The user view of the Vx960 I/O system is source-compatible with that of the UNIX I/O system. Standard C language libraries for both basic and buffered I/O functions are provided. Internally, however, the I/O system has a design that makes the Vx960 I/O system faster and more flexible than the UNIX I/O system and most other I/O systems. These are important attributes in a real-time system.

This chapter discusses the following:

- the nature of *files* and *devices*
- the user view of basic and buffered I/O

- the details of some specific devices
- the internal structure of the Vx960 I/O system.

4.2 Files, Devices, and Drivers

In Vx960, as in UNIX, applications access I/O devices by opening named *files*. A *file* can refer to one of two things:

- an unstructured *"raw" device* such as a serial communications channel or an intertask pipe
- a *logical file* on a structured, random-access device containing a file system.

Consider the following named files:

```
/usr/myfile        /pipe/mypipe        /tyCo/0
```

The first refers to a file called "myfile", on a disk device "/usr". The second is a named pipe (by convention, pipe names begin with "/pipe"). The third refers to a physical serial channel. However, I/O can be done to or from any of these in the same way. Within Vx960, they are called *files*, even though they refer to different physical objects.

Devices are handled by program modules called *drivers*. In general, using the I/O system does not require any further understanding of the implementation of devices and drivers. However, the Vx960 I/O system gives drivers flexibility in the way they handle specific devices. Drivers strive to follow the conventional user view presented here, but can differ in the specifics. See the section below, **4.6 Devices in Vx960**.

Although all I/O is directed at named files, it can be done at two different levels: *basic* and *buffered*. The two differ in the way data is buffered and in the types of calls that you can make. These two levels are discussed in later sections.

4.2.1 File Names and the Default Device

A file name is specified as a character string. An unstructured device is specified with the device name. In the case of file system devices, the device name is followed

by a file name. Thus the name "/ty0" might name a particular serial I/O channel, and the name "DEV1:/file1" would indicate the file "file1" on the device "DEV1:".

When a file name is specified in an I/O call, the I/O system searches for a device whose name matches at least an initial substring of the file name. The I/O function is then directed at this device.

If a matching device name cannot be found, then the I/O function is directed at a *default device*. You can set this default device to be any device in the system, including no device at all, in which case failure to match a device name returns an error.

Non-block devices are named when they are added to the I/O system, at system initialization time. Block devices are named when they are initialized for use with a specific file system. The Vx960 I/O system imposes no restrictions on the names given to devices. The I/O system does not interpret device or file names in any way, other than during the search for matching device and file names.

It is useful to adopt some naming conventions for device and file names. Conventions used in Vx960 give the names a UNIX look, although the semantics are somewhat different. Most device names begin with a slash ("/"), except non-NFS network devices and DOS file system devices.

By convention, NFS-based network devices are *mounted* with names that begin with a slash; for example:

```
/usr
```

Non-NFS network devices are named with the remote machine name followed by a colon; for example:

```
host:
```

The remainder of the name is the file name in the remote directory on the remote system.

File system devices using DOS are often named with uppercase letters and/or digits followed by a colon; for example:

```
DEV1:
```

Files in DOS file systems are often separated by backslashes ("\"). These can be used interchangeably with forward slashes ("/").

4.3 Basic I/O

Basic I/O is the lowest level of I/O in Vx960. The basic I/O interface is source compatible with the I/O primitives in the standard C library. There are seven basic I/O calls, shown in the following table.

Table 4-1. Basic I/O Routines

Call	Description
creat( )	Create a file.
delete( )	Delete a file.
open( )	Open a file. (Optionally, create a file.)
close( )	Close a file.
read( )	Read a created or opened file.
write( )	Write a created or opened file.
ioctl( )	Perform special control functions on files or devices.

4.3.1 File Descriptors

At the basic I/O level, files are referred to by a *file descriptor*, or *fd*. An *fd* is a small integer returned by a call to *open*( ) or *creat*( ). The other basic I/O calls take an *fd* as a parameter to specify the intended file. An *fd* has no meaning discernible to you. It is a handle for the I/O system.

When a file is opened, an *fd* is allocated and returned. When the file is closed, the *fd* is deallocated. There is a finite number of *fd*s available within the Vx960 system. To avoid exceeding the system limit, it is important to close *fd*s that are no longer in use. The number of available *fd*s is specified in the initialization of the I/O system. For more information, see the section below, **4.7 Initializing the I/O System**, and **7. Cross-Development**.

4.3.2 Standard Input, Standard Output, Standard Error

Three *fd*s are reserved and have special meanings:

0 = standard input
1 = standard output
2 = standard error output

These *fd*s are never returned as the result of an *open*( ) or *creat*( ), but serve as indirect references that can be redirected to any other open *fd*.

These standard *fd*s are used to make tasks and modules independent of their actual I/O assignments. If a module sends its output to standard output (*fd* = 1), then its output can be redirected to any file or device without altering the module.

Vx960 allows two levels of redirection. First, there is a global assignment of the three standard *fd*s. Second, individual tasks can override the global assignment of these *fd*s with their own assignments that apply only to that task.

4.3.2.1 Global Redirection

When Vx960 is initialized, the global assignments of the standard *fd*s are directed, by default, to the system console. When tasks are spawned, they have no task-specific assignments, but instead use the global assignments.

The global assignments can be redirected using *ioGlobalStdSet*( ). The parameters to this routine are the global standard *fd* to be redirected, and the *fd* it should be directed to. For example, the following call sets global standard output (*fd* = 1) to be the open file whose *fd* is *fileFd*:

```
ioGlobalStdSet (1, fileFd );
```

All tasks in the system that do not have their own task-specific redirection write standard output to that file.

The Vx960 shell operators "<" and ">" allow the assignments of the global standard input and output to be redirected for the duration of a shell command. See **9. Shell** for more information.

4.3.2.2 Task-Specific Redirection

The assignments for a specific task can be redirected using the routine *ioTaskStdSet( )*. The parameters to this routine are the task ID (0 = self) whose assignment is to be redirected, the standard *fd* to be redirected, and the *fd* to which it should be directed. For example, a task making the following call writes standard output to *fileFd*:

```
ioTaskStdSet (0, 1, fileFd);
```

All other tasks are unaffected by this redirection, and subsequent global redirections of standard output do not affect this task.

4.3.3 Open and Close

Before I/O can be performed to a device, an *fd* must be opened to that device by invoking the *open( )* routine (or *creat( )*, as discussed in the next section). The arguments to *open( )* are the file name, the type of access, and, when necessary, the "mode":

```
fd = open ("name", flag, mode);
```

The possible access flags are shown in the following table.

Table 4-2. File Access Flags

Flag			Description
O_RDONLY	or	READ	Open for reading only.
O_WRONLY	or	WRITE	Open for writing only.
O_RDWR	or	UPDATE	Open for reading and writing.
O_CREAT			Create a new file.

The *mode* parameter is used in the following special cases to specify the UNIX mode of a file or create subdirectories.

In general, only preexisting devices and files can be opened with *open( )*. However, with NFS network devices and with DOS and RT-11 file system devices, files can also be created with *open( )* by combining O_CREAT with one of the access flags. In the case of NFS devices, *open( )* requires the third parameter specifying the UNIX mode of the file:

```
fd = open ("name", O_CREAT | O_RDWR, 0644);
```

With both DOS and NFS devices, the O_CREAT option can also be used to create a subdirectory by setting *mode* to FSTAT_DIR. Other uses of the mode parameter with DOS devices are ignored.

The *open( )* routine, if successful, returns an *fd* (a small integer). This *fd* is then used in subsequent I/O calls to specify that file.

The *fd* is a *global* identifier that is *not* task specific. One task can open a file, and then any other tasks can use the resulting *fd*.

The *fd* remains valid until *close( )* is invoked with that *fd*:

```
close (fd);
```

At that point, I/O to the file is flushed (written out) and the *fd* can no longer be used by any task. However, the same *fd* number can be assigned again by the I/O system in any subsequent *open( )*.

When a task exits or is deleted, the files opened by that task are not automatically closed, since *fd*s are not task specific. Tasks should close all files when they are no longer needed. There is a limit to the number of files that can be open at once.

4.3.4 Create and Delete

File-oriented devices need to be able to create and delete files as well as open existing files. The *creat( )* routine directs a file-oriented device to make a new file on the device and return a file descriptor for it. The arguments to *creat( )* are similar to those of *open( )* except that the file name specifies the name of the new file rather than an existing one:

```
fd = creat ("name", flag);
```

The *creat( )* routine returns an *fd* identifying the new file.

The *delete( )* routine deletes a named file on a file-oriented device:

```
delete ("name");
```

Files should not be deleted while they are open.

With non-file-system oriented device names, *creat( )* acts like *open( )*; however, *delete( )* has no effect.

4.3.5 Read and Write

Once an *fd* has been obtained by invoking *open( )* or *creat( )*, tasks can read bytes from a file with *read( )* and write bytes to a file with *write( )*. The arguments to *read( )* are the *fd*, the address of the buffer to receive the input, and the maximum number of bytes to read:

```
nBytes = read (fd, &buffer, maxBytes);
```

The *read( )* routine waits for input to be available from the specified file and returns the number of bytes read. For file-system devices, if the number of bytes read is less than the number requested, a subsequent *read( )* should return 0, indicating end-of-file. For non-file-system devices, the number of bytes read can be less than the number requested even if more bytes are available; a subsequent *read( )* may or may not return 0. In this case, repeated calls to *read( )* may be necessary to read a specific number of bytes. (See the manual entry for *fioRead( )* in fioLib(1)). A return value of ERROR (-1) indicates an unsuccessful read.

The arguments to *write( )* are the *fd*, the address of the buffer that contains the data to be output, and the number of bytes to be written:

```
actualBytes = write (fd, &buffer, nBytes);
```

The *write( )* routine ensures that all specified data is at least queued for output before returning to the caller, though the data may not have been written to the device (this is driver dependent). The *write( )* routine returns the number of bytes written; if the number returned is not equal to the number requested, an error has occurred.

4.3.6 I/O Control

The *ioctl( )* routine is an open-ended mechanism for performing any I/O functions that do not fit the other basic I/O calls. Examples include determining how many bytes are available for input, setting device-specific options, obtaining information about a file system, and positioning random-access files to specific byte positions. The arguments to the *ioctl( )* routine are the *fd*, a code that identifies the specific control function requested, and an optional function-dependent argument:

```
result = ioctl (fd, function, arg);
```

For example, the call:

```
status = ioctl (fd, FIOBAUDRATE, 9600);
```

uses the FIOBAUDRATE function to the set baud rate of a *tty* device to 9600.

The discussions of the specific devices in the section below, **4.6 Devices in Vx960**, summarize the *ioctl( )* functions available for each device. The specific *ioctl( )* control codes are defined in the header file **ioLib.h**. See **Drivers(2)**, of the *Vx960 Reference Manual* for information on individual device drivers.

4.4 Buffered I/O: Stdio

The Vx960 "standard I/O library," **stdioLib**(1), provides a complete, buffered I/O package compatible with the *stdio* package in UNIX. The package includes routines for opening buffered files, getting characters from a file and putting characters into a file, formatting input and output, and other utility functions. For a task to use the Vx960 *stdio* facility, it must be spawned with the *stdio* option bit **VX_STDIO** set in the task option word.

The implementation of *printf( )* and *sscanf( )*, considered part of the *stdio* package, is different in Vx960. This is discussed in the section below, **4.5 Other Formatted I/O**.

4.4.1 Using Stdio

Although the Vx960 I/O system is efficient, some overhead is associated with each low-level call to the I/O system. First, the I/O system must dispatch from the device-independent user call (*read*( ), *write*( ), etc.) to the driver's specific routine for that function. Then, most drivers invoke some mutual exclusion or queuing mechanism, such as semaphores, to prevent simultaneous requests by multiple users from interfering with each other.

Because the Vx960 primitives are fast, this overhead is small. However, an application processing a character at a time from a file would incur that overhead for each character if it were to read each character with the basic I/O *read*( ) function:

```
n = read (fd, &char, 1);
```

To make this sort of I/O more efficient and flexible, the *stdio* package implements a buffering scheme in which data is read and written in large chunks and buffered privately. This buffering is transparent to the application; it is handled by the *stdio* routines and macros.

To access a file with *stdio*, a file is opened with *fopen*( ) instead of *open*( ) (many *stdio* calls begin with "f"):

```
fp = fopen ("/usr/foo", O_RDONLY);
```

The returned value, a *file pointer*, or *fp*, is a handle for the opened file and its associated buffers and pointers. An *fp* is a pointer to the associated data structure of type FILE (i.e., it is declared as "FILE *"). By contrast, the low-level I/O routines identify a file with a *file descriptor* (*fd*) which is a small integer. In fact, the FILE structure pointed to by the *fp* contains the underlying *fd* of the open file.

An already open *fd* can be associated with a FILE buffer by calling *fdopen*( ):

```
fp = fdopen (fd, O_RDONLY);
```

After a file has been opened with *fopen*( ), data can be read with *fread*( ), or a character at a time with *getc*( ), and data can be written with *fwrite*( ), or a character at a time with *putc*( ).

The routines and macros to get data into or out of a file are efficient. They access the buffer via direct pointers that are incremented as you read or write data. They pause to call the low-level read or write functions only when a read buffer is empty or a write buffer is full.

Note: The *stdio* buffers and pointers are *private* to a particular task. They are *not* interlocked with semaphores or any other mutual exclusion mechanism, as this would defeat the point of an efficient private buffering scheme. Therefore, multiple tasks must not perform I/O to the same *stdio* FILE pointer at the same time.

4.4.2 Standard Input, Standard Output, Standard Error

As discussed earlier in section **4.3 Basic I/O**, there are three special file descriptors (0, 1, and 2) reserved for standard input, standard output, and standard error. When a task is spawned with the VX_STDIO option bit-set, three corresponding *stdio* FILE buffers are automatically created and associated with those file descriptors: *stdin*, *stdout*, and *stderr*. These can be used, as in UNIX, to do buffered I/O to the standard *fd*s. For more information, see **3.2.4.4 Task Options**.

4.5 Other Formatted I/O

4.5.1 Special Cases: *printf*( ), *sprintf*( ), and *sscanf*( )

The routines *printf*( ), *sprintf*( ), and *sscanf*( ) are considered to be part of the standard *stdio* package. However, the Vx960 implementation of these routines, while the same, does not use the *stdio* package. Instead, it uses a self-contained, formatted, but unbuffered interface to the I/O system in the library fioLib(1).

There are two principal reasons for implementing these routines in this way. Without sacrificing their availability, (1) a task can be spawned without the VX_STDIO option; and (2) the entire *stdio* package, which is optional, can be omitted from a Vx960 configuration. Applications requiring *printf*-style output which is buffered can still accomplish this by calling *fprintf*( ) to *stdout*.

While *sscanf*( ) is implemented in **fioLib** and can be used even if *stdio* is omitted, the same is not true of the routine *scanf*( ), which is implemented in the usual way in *stdio*.

4.5.2 Additional Routines: *printErr( )* and *fdprintf( )*

There are additional routines in fioLib that provide formatted but unbuffered output. The routine *printErr( )* is analogous to *printf( )* but outputs formatted strings to the standard error *fd* (2). The routine *fdprintf( )* outputs formatted strings to a specified *fd*.

4.5.3 Message Logging

Another higher-level I/O facility is provided by the library logLib(1), which allows formatted messages to be logged without having to do I/O in the current task's context, or when there is no task context. The message format and parameters are sent on a message queue to a logging task, which then formats and outputs the message. This is useful when messages must be logged from interrupt level, or when you do not want to delay the current task for I/O or you use the current task's stack for message formatting (formatting can take up significant stack space). The message is displayed on the console unless otherwise redirected by *logInit( )* or *logFdSet( )*.

4.6 Devices in Vx960

The Vx960 I/O system allows considerable flexibility in the way specific device drivers handle the seven I/O functions. All device drivers follow the basic conventions outlined above, but differ in specifics. The following sections describe the nature of I/O for each of the devices supplied with Vx960.

4.6.1 Supplied Drivers

The following drivers are provided with Vx960:

Table 4-3. Vx960 Drivers

Module	Driver Description
tyCoDrv(2)	Terminal Driver (target specific)
ptyDrv(2)	Pseudo-terminal Driver
pipeDrv(2)	Pipe Driver
netDrv(2)	Network Driver
nfsDrv(2)	NFS Driver
ramDrv(2)	RAM Driver
scsiLib(1)	SCSI interface library
-	Other hardware-specific drivers

4.6.2 Serial I/O Devices (Terminal and Pseudo-Terminal Devices)

Vx960 provides terminal and pseudo-terminal device drivers (*tty* and *pty* drivers). The *tty* driver is used for actual terminals. The *pty* driver is used for processes which simulate terminals. Pseudo terminals are useful in applications such as remote login facilities[1].

Vx960 serial I/O devices are buffered serial byte streams. Each device has a ring buffer (circular buffer) for both input and output. Reading from a *tty* device extracts bytes from the input ring. Writing to a *tty* device adds bytes to the output ring. The size of each ring buffer is specified when the device is created during system initialization.

1. For the remainder of this section, the term *tty* is used to indicate both *tty* and *pty* devices.

4.6.2.1 Tty Options

The *tty* devices have a full range of options that affect the behavior of the device. These options are selected by setting bits in the device option word using the *ioctl( )* routine with the FIOSETOPTIONS function (see **4.6.2.4 I/O Control Functions** below). For example:

```
status = ioctl (fd, FIOSETOPTIONS, OPT_TERMINAL);
```

Table 4-4 below is a summary of the available options. The listed names are defined in the header file **ioLib.h**. For more detailed information see the manual entry for **ioLib**(1).

Table 4-4. Tty Options

Library	Description
OPT_LINE	Select *line mode*. (See the following section.)
OPT_ECHO	Echo input characters to the output of the same channel.
OPT_CRMOD	Translate input RETURN characters into NEWLINEs (\n); translate output NEWLINEs into RETURN-LINEFEEDs.
OPT_TANDEM	Respond to X-on/X-off protocol (^Q and ^S).
OPT_7_BIT	Strip the most significant bit from all input bytes.
OPT_MON_TRAP	Enable the special *ROM monitor trap* character, ^X by default.
OPT_ABORT	Enable the special *shell abort* character, ^C by default.
OPT_TERMINAL	Set all of the above option bits.
OPT_RAW	Set none of the above option bits.

4.6.2.2 Raw Mode and Line Mode

A *tty* device operates in one of two modes: *raw mode* (unbuffered) or *line mode*. Raw mode is the default mode. Line mode is selected by the OPT_LINE bit of the device option word (see the previous section).

In *raw mode,* each input character is available to readers as soon as it is input from the device. Reading from a *tty* device in raw mode causes as many characters as possible to be extracted from the input ring, up to the limit of the user's read buffer. Input cannot be modified except as directed by other *tty* option bits.

In *line mode,* all input characters are saved until a NEWLINE character is input; then the entire line of characters, including the NEWLINE, is made available in the ring all at once. Reading from a *tty* device in line mode causes characters up to the end of the next line to be extracted from the input ring, up to the limit of the user's read buffer. Input can be modified by the special characters ^H (backspace), ^U (line-delete), and ^D (end-of-file) discussed in the following section.

4.6.2.3 Tty Special Characters

The following special characters are enabled if the *tty* device operates in line mode, i.e., with the OPT_LINE bit set:

- The backspace character, by default ^H, causes successive previous characters to be deleted from the current line, up to the start of the line. It does this by echoing a backspace followed by a space, and then another backspace.
- The line-delete character, by default ^U, deletes all the characters of the current line.
- The end-of-file (EOF) character, by default ^D, causes the current line to become available in the input ring without a NEWLINE and without entering the EOF character itself. Thus if the EOF character is the first character typed on a line, reading that line returns a zero byte count, which is the usual indication of end-of-file.

The following special characters are enabled if the *tty* device operates with their respective option bits set:

- The flow control characters, ^Q and ^S, in what is known as X-on/X-off protocol. Receipt of a ^S input character suspends output to that channel. Subsequent receipt of a ^Q resumes the output. Also, when the Vx960 input buffer is almost full, a ^S is output to signal the other side to suspend transmission. When the input buffer is empty enough, a ^Q is output to signal the other side to resume transmission. X-on/X-off protocol is enabled by OPT_TANDEM.
- The *ROM monitor trap* character, by default ^X, faults to the ROM resident monitor program. Vx960 functioning is suspended, and the computer system is con-

trolled by the monitor. Depending on the particular monitor, it may or may not be possible to restart Vx960 from the point of interruption. The monitor fault character is enabled by OPT_MON_TRAP.

- Enables the special *shell abort* character, by default ^C. When this character is received and this option is enabled, the Vx960 shell is restarted. This is useful for freeing a shell caught in an infinite loop or one that has taken an unavailable semaphore. For more information see **9. Shell**. The shell abort character is enabled by OPT_ABORT.

The characters for most of these functions can be changed using the tyLib(1) routines shown in Table 4-5 below.

Table 4-5. Tty Special Characters

Character	Description	Modifier
^H	backspace (character delete)	*tyBackspaceSet( )*
^U	line delete	*tyDeleteLineSet( )*
^D	EOF (end of file)	*tyEOFSet( )*
^C	shell abort	*tyAbortSet( )*
^X	ROM monitor trap	*tyMonitorTrapSet( )*
^S	output suspend	N/A
^Q	output resume	N/A

4.6.2.4 I/O Control Functions

The *tty* devices respond to the *ioctl( )* functions summarized in the following table. The functions listed are defined in the header **ioLib.h**. For more information, see the manual entry for tyLib(1) and the manual entry for *ioctl( )* in ioLib(1).

Table 4-6. I/O Control Functions Supported by tyLib

Function	Meaning
FIOGETNAME	Get the file name of the *fd*.
FIOSETOPTIONS	Set the device option word.
FIOGETOPTIONS	Return the current device option word.
FIONREAD	Get the number of unread bytes in the input buffer.
FIONWRITE	Get the number of bytes in the output buffer.
FIOFLUSH	Discard all bytes in the input and output buffers.
FIOCANCEL	Cancel a read or write.
FIOBAUDRATE	Set the baud rate to the specified argument.

4.6.3 Pipe Devices

Pipes are virtual devices by which tasks communicate with each other through the I/O system. Tasks write messages to pipes which can then be read by other tasks. Pipe devices are managed by **pipeDrv** and use the kernel message queue facility to bear the actual message traffic.

4.6.3.1 Creating Pipes

Pipes are created by calling the pipe create routine:

```
status = pipeDevCreate ("/pipe/name", maxMsgs, maxLength );
```

The new pipe is able to have at most *maxMsgs* messages queued at a time. Tasks that write to a pipe that already has the maximum number of messages queued are delayed until room is made available by the dequeuing of a message. Each message in the pipe can be at most *maxLength* bytes long. Attempts to write messages longer than the maximum result in an error.

4.6.3.2 Writing to Pipes from Interrupt Service Routines

Vx960 pipes are designed to make it possible for interrupt service routines to write to pipes in the same way as task-level code. Many Vx960 facilities are not available to interrupt service routines, including I/O to devices other than pipes. However, interrupt service routines can use pipes to communicate with tasks which can then invoke such facilities.

Interrupt service routines write to a pipe using the normal *write( )* call. Tasks and interrupt routines can write to the same pipes. However, if the pipe is full, the message written is discarded since the interrupt service routines cannot pend. Interrupt service routines must not invoke any I/O function on pipes other than *write( )*.

4.6.3.3 I/O Control Functions

Pipe devices respond to the *ioctl( )* functions summarized in the following table. The functions listed are defined in the header **ioLib.h**. For more information, see the manual entry for pipeDrv(2) and the manual entry for *ioctl( )* in ioLib(1).

Table 4-7. I/O Control Functions Supported by pipeDrv

Function	Meaning
FIOGETNAME	Get the file name of the *fd*.
FIONREAD	Get the number of unread bytes in the pipe.
FIOFLUSH	Discard all messages in the pipe.
FIONMSGS	Get the number of messages remaining in the pipe.

4.6.4 Network File System (NFS) Devices

NFS devices allow files on remote hosts to be accessed via the NFS protocol. The driver **nfsDrv** acts as an NFS *client* to access files on any NFS *server* on the network. Using NFS devices, remote files are created, opened, and accessed as though they were on a file system on a local disk. This is called *network transparency*.

4.6.4.1 Mounting an NFS File System

Access to a remote NFS file system is established by mounting that file system locally and creating an I/O device for it using *nfsMount( )*. Its arguments are (1) the host name of the NFS server, (2) the name of the host file system, and (3) the local name for the file system.

For example, the following call mounts /usr of the host mars as /vxusr locally:

```
nfsMount ("mars", "/usr", "/vxusr");
```

This creates a Vx960 I/O device with the specified local name (i.e., /vxusr in this example). If the local name is specified NULL, the local name is the same as the remote name.

After a remote file system is mounted, the files are accessed as though the file system were local. Thus, after the example above, opening the file /vxusr/foo opens the file /usr/foo on the host mars.

The remote file system must have been *exported* by the system on which it resides. Also NFS requires *authentication* parameters to identify the user making the remote access. These parameters are set using the routines *nfsAuthUnixSet( )* and *nfsAuthUnixPrompt( )*.

The issues of exporting and mounting NFS file systems, and authenticating access permissions, are discussed in more detail in **6.3.3.2 Transparent Remote File Access via NFS**. See also the manual entries nfsLib(1) and nfsDrv(2), and the NFS documentation from Sun Microsystems.

4.6.4.2 I/O Control Functions

NFS devices respond to the *ioctl( )* functions summarized in the following table. The functions listed are defined in the header ioLib.h. For more information, see the manual entry for nfsDrv(2) and the manual entry for *ioctl( )* in ioLib(1).

Table 4-8. I/O Control Functions Supported by nfsDrv	
Function	**Meaning**
FIOGETNAME	Get the file name of the *fd*.
FIONREAD	Get the number of unread bytes in the file.
FIOSEEK	Set the current byte offset in the file.
FIOWHERE	Return the current byte position in the file.
FIOREADDIR	Read the next directory entry.
FIOFSTATGET	Get file status information (directory entry data).

4.6.5 Non-NFS Network Devices

Vx960 supports network access to files on the remote host via the UNIX remote shell, *rsh*, or the File Transfer Protocol, *ftp*. These implementations of network devices use the driver netDrv(2). When a remote file is opened using *rsh* or *ftp*, the entire file is copied into local memory. Subsequent read and write operations are performed on the in-memory copy of the file. When closed, the file is copied back to the original remote file if it has been modified.

In general, NFS devices are preferable to *rsh* and *ftp* devices for performance and flexibility, since NFS does not copy the entire file into local memory. However, NFS is more cumbersome to administer and is not supported by all host systems.

4.6.5.1 Creating Network Devices

To access files on a remote host using either *rsh* or *ftp*, a network device must first be created by calling the routine *netDevCreate( )*. The arguments to *netDevCreate( )* are (1) the name of the device, (2) the name of the host the device accesses and (3) which protocol is used: 0 (*rsh*) or 1(*ftp*).

For example, the following call creates an *rsh* device called "mars:" that accesses the host "mars". By convention, a network device's name is the remote machine's name followed by a colon (":").

```
netDevCreate ("mars:", "mars", 0);
```

Files on a network device can be created, opened, and manipulated just like files on a local disk. Thus, opening the file **mars:/usr/foo** would open **/usr/foo** on host **mars**.

Creating a network device allows access to any file or device on the remote system, while mounting an NFS file system allows access only to a specified file system.

For the files of a remote host to be accessible via *rsh* or *ftp*, permissions and user identification must be established on both the remote and local systems. Creating and configuring network devices is discussed in more detail in **6.3.3.1 Transparent Remote File Access via rsh and ftp**. See also the manual entry **netDrv**(2).

4.6.5.2 I/O Control Functions

Rsh and *ftp* devices respond to the same *ioctl*( ) requests as NFS devices. The functions are defined in the header **ioLib.h**. For more information, see the manual entry for netDrv(2) and the manual entry for *ioctl*( ) in **ioLib**(1).

4.6.6 Block Devices (Disk and RAM Disk Devices)

A *block device* is a device that is organized as a sequence of individually accessible blocks of data. The most common type of block device is a disk.

When discussing a block device in Vx960, the term *block* refers to the smallest addressable unit on the device. For most disk devices, a Vx960 block corresponds to a *sector*, although terminology varies.

Block devices in Vx960 have a different interface than other I/O devices. Rather than interacting with the I/O system, block device drivers instead interact with a file system. The file system, in turn, interacts with the I/O system. This arrangement allows a single block device driver to be used with various different file systems and reduces the number of I/O functions that must be supported in the driver. The internal implementation of block device drivers is discussed in section **4.9.4 Block Devices**.

4.6.6.1 File Systems

For use with block devices, Vx960 is supplied with file system libraries compatible with the MS-DOS and RT-11 file systems. Also supplied is a library for a simple "raw" disk file system, which treats an entire disk much like a single large file. Use of these file systems is discussed in **5. Local File Systems**. Also see the manuals entries for **dosFsLib(1)**, **rt11FsLib(1)**, and **rawFsLib(1)**.

4.6.6.2 SCSI Devices

SCSI is a standard peripheral interface that allows connection with a wide variety of Winchester disks, optical disks, floppy disks, and tape drives. SCSI block devices are compatible with the DOS and RT-11 file system libraries, and offer several advantages for target configurations. They provide:

- local mass storage in non-networked environments
- faster and more deterministic I/O throughput than Ethernet networks.

The Vx960 SCSI implementation consists of two modules. The routines that support the device-independent SCSI interface are found in **scsiLib**. Libraries of configuration routines exist that support specific SCSI controllers (e.g., **wd33c93Lib** for the Western Digital WD33C93 device or **mb87030Lib** for the Fujitsu MB87030 device). Additional support routines exist for individual targets in **sysLib.c**.

- **Configuring SCSI Devices**

The SCSI interface is included in the Vx960 system image by adding the line:

```
#define INCLUDE_SCSI
```

to the file config.h. After Vx960 is built, the include file initializes the SCSI interface by executing *sysScsiInit*( ) and *usrScsiConfig*( ).

The routine *sysScsiInit*( ) initializes the SCSI controller and sets up interrupt handling. The physical device configuration is specified in the routine *usrScsiConfig*( ) which is found in **usrConfig.c**. The routine contains an example of the calling sequence to declare a hypothetical configuration, including the definition of physical devices with *scsiPhysDevCreate*( ), the creation of logical partitions with *scsiBlkDevCreate*( ), and the specification of a file system with either *dosFsDevInit*( ) or *rt11FsDevInit*( ).

Modify the routine *usrScsiConfig( )* to reflect your actual configuration. For more information on the routines used in *usrScsiConfig( )*, see the manual entries for *scsiPhysDevCreate( )*, *scsiBlkDevCreate( )*, *dosFsDevInit( )*, *rt11FsDevInit( )*, *dosFsMkfs( )*, and *rt11FsMkfs( )*.

If problems with your configuration occur, insert the following lines at the beginning of *usrScsiConfig( )*:

```
scsiDebug = TRUE;

scsiIntsDebug = TRUE;
```

to obtain further information on SCSI bus activity. Do not declare the global variables *scsiDebug* and *scsiIntsDebug* locally. You can set or reset them from the Vx960 shell.

4.6.6.3 RAM Disk Devices

RAM devices, as implemented in **ramDrv**, emulate disk devices but keep all data in memory. Memory location and "disk" size are specified when a RAM device is created by calling *ramDevCreate( )*. You can call this routine repeatedly to create multiple RAM disks.

Memory for the RAM disk can be preallocated and the address passed to *ramDevCreate( )*, or memory can be allocated from the system memory pool using *malloc( )*.

Once the device has been created, a name and file system (DOS, RT-11, or raw disk) must be associated with it using the file system's device initialization routine or make-file-system routine, e.g., *dosFsDevInit( )* or *dosFsMkfs( )*. Information describing the device is passed to the file system via a BLK_DEV structure. The RAM disk creation routine returns a pointer to this structure.

In the following example, a 200-Kbyte RAM disk is created with allocated memory, 512-byte sections, a single track, and no sector offset. The device is then initialized for use with DOS and assigned the name "DEV1:".

```
BLK_DEV       *pBlkDev;
DOS_VOL_DESC  *pVolDesc;

pBlkDev = ramDevCreate (0, 512, 400, 400, 0);
pVolDesc = dosFsMkfs ("DEV1:", pBlkDev);
```

The *dosFsMkfs*( ) routine calls *dosFsDevInit*( ) with default parameters and initializes the file system on the disk by calling *ioctl*( ) with the FIODISKINIT function.

If the RAM disk memory already contains a disk image, the first argument to *ramDevCreate*( ) is the address in memory, and the formatting arguments must be identical to those used when the image was created. For example:

```
pBlkDev = ramDevCreate (0xc0000, 512, 400, 400, 0);
pVolDesc = dosFsDevInit ("DEV1:", pBlkDev, NULL );
```

In this case, *dosFsDevInit*( ) must be used instead, since the file system already exists on the disk and should not be re-initialized. This procedure is useful if a RAM disk is to be created at the same address used in a previous boot of Vx960. The contents of the RAM disk is then preserved.

Follow these same procedures to create a RAM disk with RT-11 using *rt11FsMkfs*( ) and *rt11FsDevInit*( ).

For more information on RAM disk devices, see the manual entry for ramDrv(2). For more information on file systems, see **5. Local File Systems.**

4.6.7 Sockets

In Vx960, the underlying basis of network communications is *sockets*. A socket is an endpoint for communication between tasks; data is sent from one socket to another.

Sockets are *not* created or opened using the standard I/O functions. Instead they are created by calling *socket*( ), and connected and accessed using other routines in sockLib(1). However, once you create and connect a *stream* socket (using TCP), you can access it as a standard I/O device, using *read*( ), *write*( ), *ioctl*( ), and *close*( ). The value returned by *socket*( ) as the socket handle is an I/O system *fd*.

These routines are source compatible with the socket functions of BSD 4.3 UNIX. Use of these routines is discussed in **6.2.5 Sockets.**

4.7 Initializing the I/O System

Before using the I/O system, you must initialize it by calling *iosInit( )*. Call *iosInit( )* in the root task in usrConfig. You cannot call I/O functions until you have initialized the I/O system.

The arguments to *iosInit( )* are the maximum number of drivers and the maximum number of open files that are allowed in the system. The I/O system allocates space for the driver table and the *fd* table large enough to accommodate the specified maximums.

4.8 Differences Between Vx960 and UNIX I/O

Most uses of I/O in Vx960 are source-compatible with I/O in UNIX. However, note the following differences:

Device Configuration
: In Vx960 device drivers can be installed and removed dynamically.

File Descriptors
: In UNIX, *fd*s are unique to each process. In Vx960, *fd*s are global entities, accessible by any task, except for standard input, standard output, and standard error (0, 1, and 2), which can be task-specific.

I/O Control
: The specific parameters passed to *ioctl( )* functions may be different in UNIX and Vx960.

4.9 Internal Structure

The Vx960 I/O system is different from most in the way the work of performing user I/O requests is partitioned between the device-independent I/O system and the individual device drivers.

In many systems, the device driver supplies a few functions to perform low-level I/O functions such as inputting or outputting a sequence of bytes to character-oriented devices. The higher-level protocols, such as communications protocols on character-oriented devices, are implemented in the device-independent part of the I/O system. The user requests are processed by the I/O system before the driver functions get control.

While this approach is designed to make it easy to implement drivers and to ensure that devices behave as much alike as possible, it has several drawbacks. The driver writer is often hampered in implementing alternative protocols that are not provided by the existing I/O system. In a real-time system, you may want to bypass the standard protocols altogether in certain devices where throughput is critical or where the device does not fit the standard model.

In the Vx960 I/O system, on the other hand, minimal processing is done on user I/O requests before control is given to the device driver. Instead, the Vx960 I/O system acts as a switch to route user requests to appropriate driver-supplied routines. Each driver can then process the raw user requests as appropriate to its devices. In addition, however, several high-level, subroutine libraries are available to driver writers that implement standard protocols for both character- and block-oriented devices. Thus the Vx960 I/O system gives you the best of both worlds: while it is easy to write a standard driver for most devices with just a few pages of device-specific code, driver writers are free to execute the user requests in nonstandard ways where appropriate. Block devices, because they must interact with Vx960 file systems, use a different organization, described in **4.9.4 Block Devices.**

As discussed earlier, the three main elements of the Vx960 I/O system are drivers, devices, and files. The following sections describe these elements in detail.

Figure 4-1 shows the abbreviated code for a hypothetical driver which is used as an example throughout the following discussions. It is typical of drivers for character-oriented devices.

```
/*******************************************************************************
* xxDrv - driver initialization routine
*
* This routine is called to initialize the driver. It installs the driver
* by calling iosDrvInstall. It may allocate data structures, connect
* interrupt routines to their interrupts, and initialize device hardware.
*/
STATUS  xxDrv ()
    {
    xxDrvNum = iosDrvInstall (xxCreat, 0, xxOpen, 0, xxRead, xxWrite, xxIoctl);
    (void) intConnect (intvec, xxInterrupt, ...);
    ...
    }
/*******************************************************************************
* xxDevCreate - device creation routine
*
* This routine is called to add a device to the system that will be
* serviced by this driver, by the specified name. Other driver-dependent
* arguments may be required such as buffer sizes, device addresses, etc.
* The routine adds the device to the I/O system by calling iosDevAdd.
* It may also allocate and initialize data structures for the device,
* initialize semaphores, initialize device hardware, etc.
*/
STATUS xxDevCreate (name, ...)
    char *name;
    ...
    {
    status = iosDevAdd (xxDev, name, xxDrvNum);
    ...
    }
/*******************************************************************************
* The following routines implement the basic I/O functions. Note that the
* return value type is driver dependent.
*/
XXDEV *xxOpen (xxDev, remainder, mode)
    XXDEV *xxDev;
    char *remainder;
    int mode;
    {
    /* serial devices should have no file name part */

    if (remainder[0] != 0)
        return (ERROR);
    else
        return (xxDev);
    }
int xxRead (xxDev, buffer, nBytes)
    XXDEV *xxDev;
    char *buffer;
    int nBytes;
    ...
int xxWrite (xxDev, buffer, nBytes)
    ...
int xxIoctl (xxDev, requestCode, arg)
    ...
/*******************************************************************************
* xxInterrupt - interrupt service routine
*
* Most drivers have routines that handle interrupts from the devices
* serviced by the driver. These routines are connected to the interrupts
* by calling intConnect (usually in xxDrv above). They can receive a
* single argument, specified in the call to intConnect (see intLib).
*/
VOID xxInterrupt (arg)
    ...
```

Figure 4-1. Example – Abbreviated Driver Code

In Vx960, each driver has a short, unique abbreviation such as "net" or "tyCo", that is prefixed to each of its routines. The abbreviation for the example driver is "*xx*".

4.9.1 Drivers

A driver for a non-block device implements the seven basic I/O functions, *creat( )*, *delete( )*, *open( )*, *close( )*, *read( )*, *write( )*, and *ioctl( )*, for a particular kind of device. In general, this type of driver has routines that implement each of these functions, although some of the routines can be omitted if the function is non-operative with that device.

A driver for a block device interfaces with a file system, rather than with the I/O system. The file system in turn implements most I/O functions. The driver need supply routines to read and write blocks, reset the device, perform I/O control, and check device status. Device drivers for block devices have a number of special requirements which are discussed in a separate section, **4.9.4 Block Devices.**

When you invoke one of the basic I/O functions, the I/O system routes the request to the appropriate routine of a specific driver, through a path that is discussed in subsequent sections. The driver's routine runs in the calling task's context, just as though it had been called from the user program. Thus, the driver is free to use any facilities available to tasks, including I/O to other devices. This means that most drivers have to use some mechanism to provide mutual-exclusion to critical regions of code. The usual mechanism is the semaphore facility provided in semLib(1).

In addition to the routines that implement the seven basic I/O functions, drivers also have three other routines:

- an initialization routine that installs the driver in the I/O system, connects to any interrupts used by the devices serviced by the driver, and performs any necessary hardware initialization (named *xxDrv( )*)
- a routine to add devices to the I/O system that are to be serviced by the driver (named *xxDevCreate( )*)
- interrupt-level routines that are connected to the interrupts of the devices serviced by the driver.

4.9.1.1 The Driver Table and Installing Drivers

The function of the I/O system is to route user I/O requests to the appropriate routine of the appropriate driver. This is done by means of a table maintained by the I/O system that contains the address of each routine for each driver. Drivers are installed by calling the I/O system internal function *iosDrvInstall( )*. The arguments to this routine are the addresses of the seven I/O routines for the new driver. The *iosDrvInstall( )* routine enters these addresses in a free slot in the driver table and returns the index of this slot. This index is known as the *driver number* and is used to associate particular devices with the driver.

Null (0) addresses can be specified for some of the seven routines. This indicates that the driver need not process those functions. For non-file system drivers, *close( )* and *delete( )* are often non-operative as far as the driver is concerned.

Vx960 file systems (dosFsLib, rt11FsLib, and rawFsLib) contain their own entries in the driver table, which are created when the file system library is initialized.

4.9.1.2 Example of Installing a Driver

Figure 4-2 shows the actions taken when our example driver calls its initialization routine *xxDrv( )*.

❶ The driver calls *iosDrvInstall( )*, specifying the addresses of the driver's routines for the seven basic I/O functions.

The I/O system:

❷ Locates the next available slot in the driver table, in this case slot 2.

❸ Enters the addresses of the driver routines in the driver table.

❹ Returns the slot number as the driver number of the installed driver.

4.9.2 Devices

A driver can be capable of servicing many instances of a particular kind of device. For example, a single driver for a serial communications device can often handle many individual channels that differ in a few parameters, such as device address.

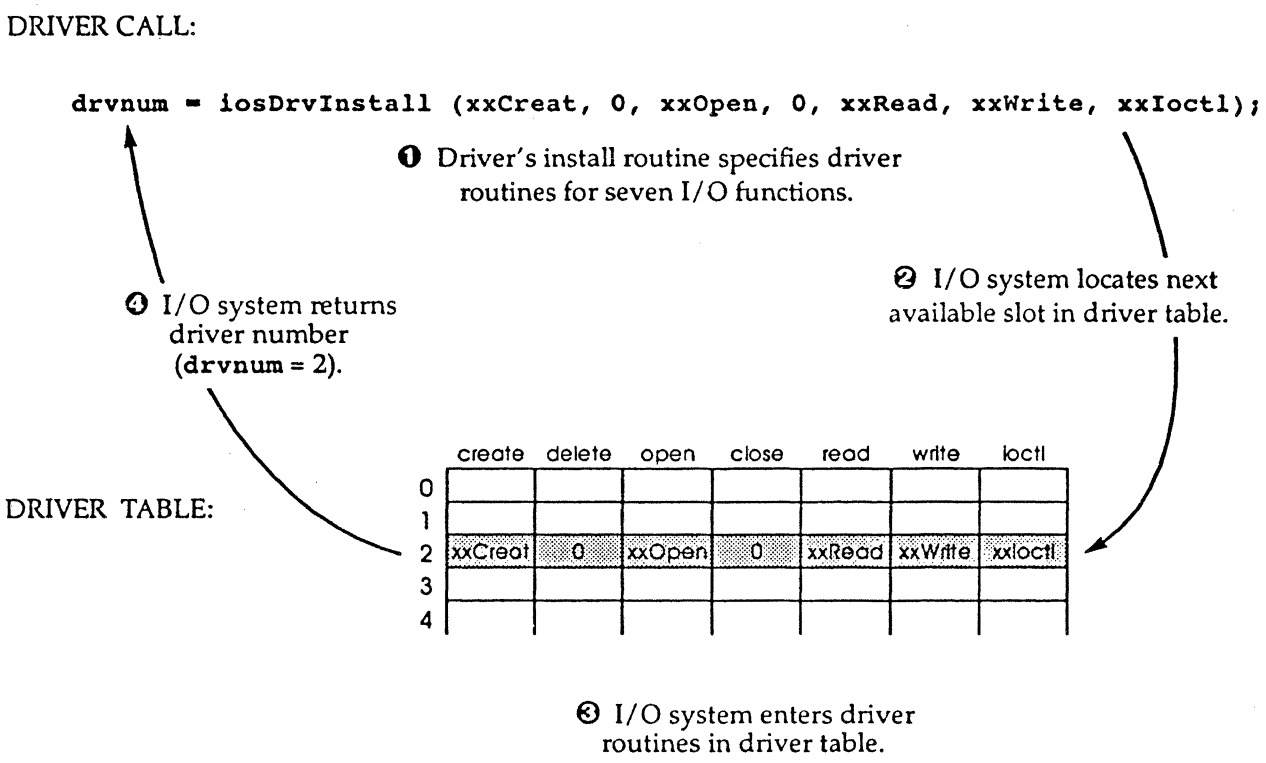


Figure 4-2. Example – Driver Initialization for Non-Block Devices

In the Vx960 I/O system, devices are defined by a data structure called a *device header* (DEV_HDR). This data structure contains the device name string and the driver number for the driver that services this device. The device headers for all the devices in the system are kept in a memory-resident linked list called the *device list*. The device header is the initial part of a larger structure determined by the individual drivers. This larger structure, called a *device descriptor*, contains additional device-specific data such as device addresses, buffers, semaphores, etc.

4.9.2.1 The Device List and Adding Devices

Non-block devices are added to the I/O system by calling the internal I/O function *iosDevAdd*( ). The arguments to *iosDevAdd*( ) are the address of the device descriptor for the new device, the device's name, and the driver number of the driver that services the device. The device descriptor specified by the driver can contain any necessary device-dependent information, as long as it begins with a device header. The driver does not need to fill in the device header, only the device-dependent information. The *iosDevAdd*( ) routine enters the specified device name and the driver number in the device header and adds it to the system device list.

Block devices are added to the I/O system by calling the device initialization routine for the file system that is needed with the device (i.e., *dosFsDevInit*( ), *rt11FsDevInit*( ), and *rawFsDevInit*( )). The file system routine then calls *iosDevAdd*( ).

4.9.2.2 Example of Adding Devices

In Figure 4-3, the device creation routine *xxDevCreate*( ) of our example driver adds devices to the I/O system by calling *iosDevAdd*( ).

4.9.3 File Descriptors

Several *fd*s can be open to a single device at one time. A device driver can maintain additional information associated with an *fd* beyond the I/O system's device information. In particular, devices on which multiple files can be open at once have file-specific information (e.g., file offset) associated with each *fd*. You can also have several *fd*s open to a non-block device, such as a *tty*. No additional information exists, and thus writing on any of the *fd*s has identical results.

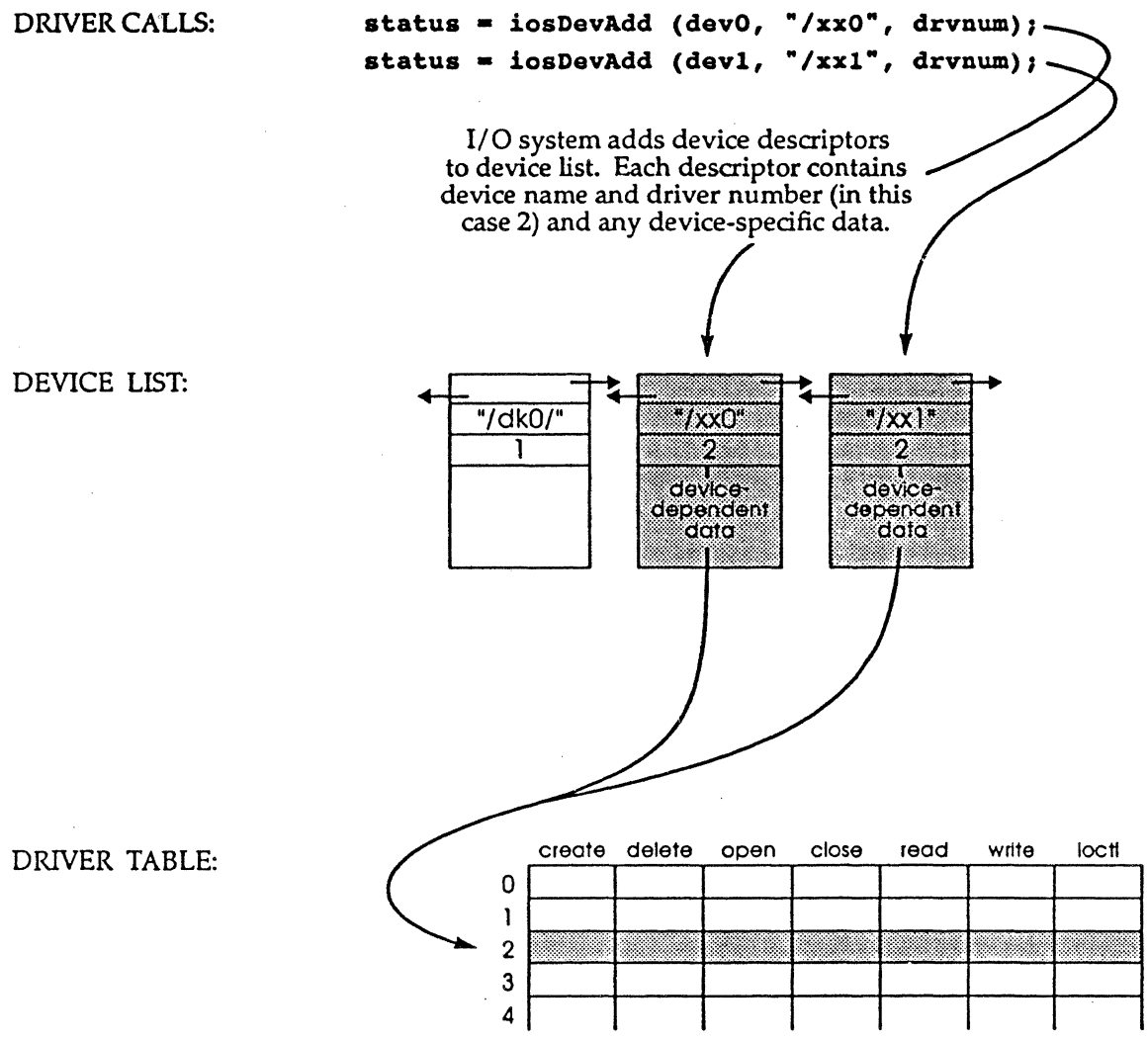


Figure 4-3. Example – Addition of Devices to I/O System

4.9.3.1 The Fd Table

Files are opened with the I/O routine *open( )* (or *creat( )*). The I/O system searches the device list for a device name that matches the file name (or an initial substring) specified by the caller. If such a match is found, the I/O system then uses the driver number contained in the corresponding device header to locate and call the driver's open routine via the driver table.

The I/O system needs to establish an association between the file descriptor that is used by the caller in subsequent I/O calls, and the driver that services it. Additionally, the driver needs to associate some data structure per descriptor. In the case of non-block devices, this is just the device descriptor that was located by the I/O system.

The I/O system maintains these associations in a table called the *fd table*. This table contains the driver number and an additional driver determined 4-byte value. The driver value is the internal descriptor returned by the driver's open routine, and can be any nonnegative value the driver requires to identify the file. In subsequent calls to the driver's other I/O functions (*read( )*, *write( )*, *ioctl( )*, and *close( )*), this value is supplied to the driver in place of the *fd* in the user's I/O call.

4.9.3.2 Example of Opening a File

In Figure 4-4 and Figure 4-5, a user calls *open( )* to open the file "/xx0". The I/O system takes the following series of actions:

1. It searches the device list for a device name that matches the specified file name, or an initial substring of the file name. In this case, a complete device name match is found.
2. It reserves a slot in the *fd* table, which is used if the open is successful.
3. It then looks up the address of the driver's open routine, *xxOpen( )*, and calls that routine. The arguments to *xxOpen( )* are transformed by the I/O system from the user's original arguments to *open( )*. The first argument to *xxOpen( )* is a pointer to the device descriptor the I/O system located in the full file name search. The next parameter is the *remainder* of the file name specified by the user, after removing the initial substring that matched the device name. In this case, since the device name matched the entire file name, the remainder passed to the driver is a null string. The driver is free to interpret this remainder in any way it wants. In the case of block devices, this remainder is the name of a file on the

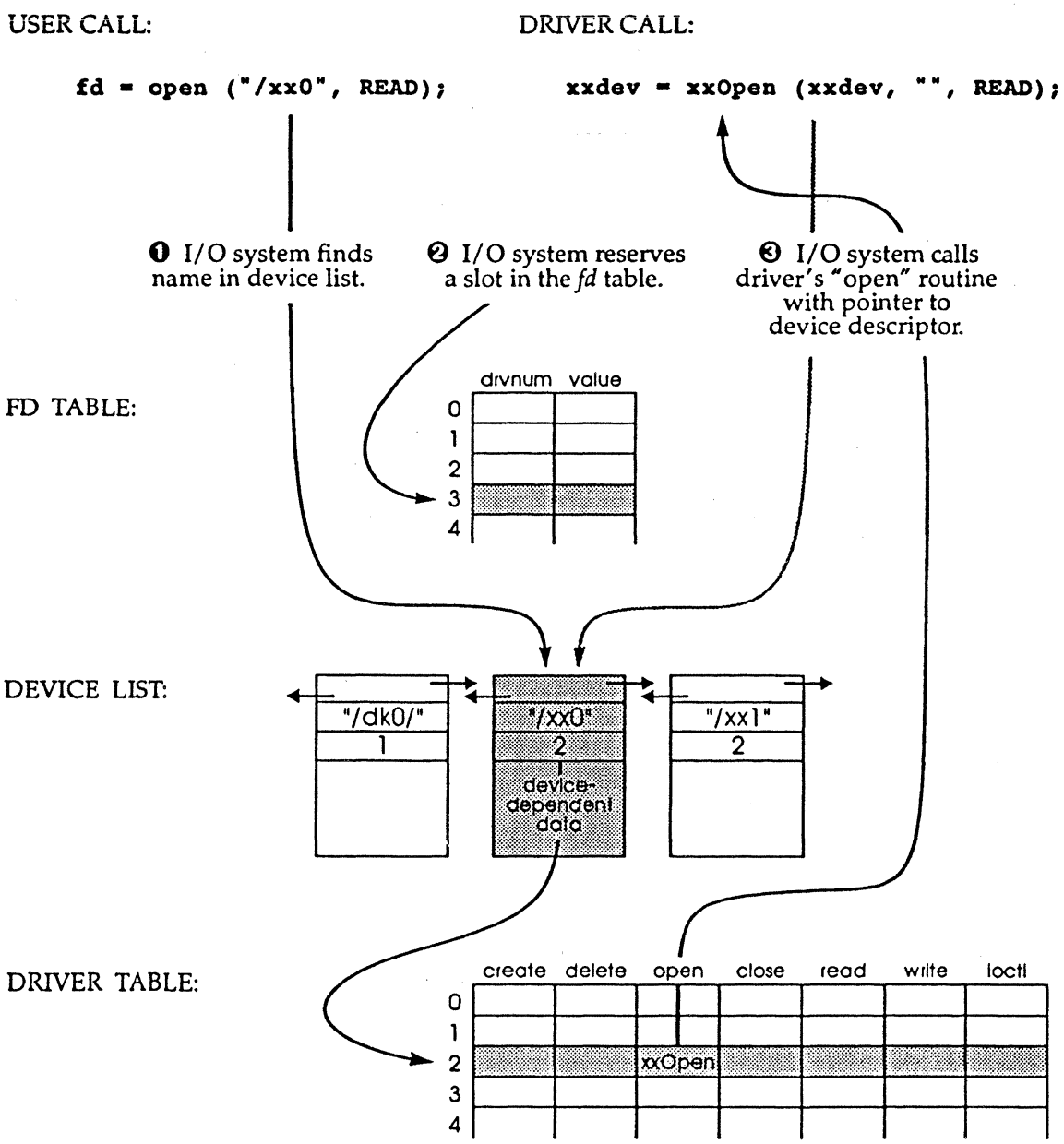


Figure 4-4. Example – Call to I/O Routine *open( )* (*Part 1*)

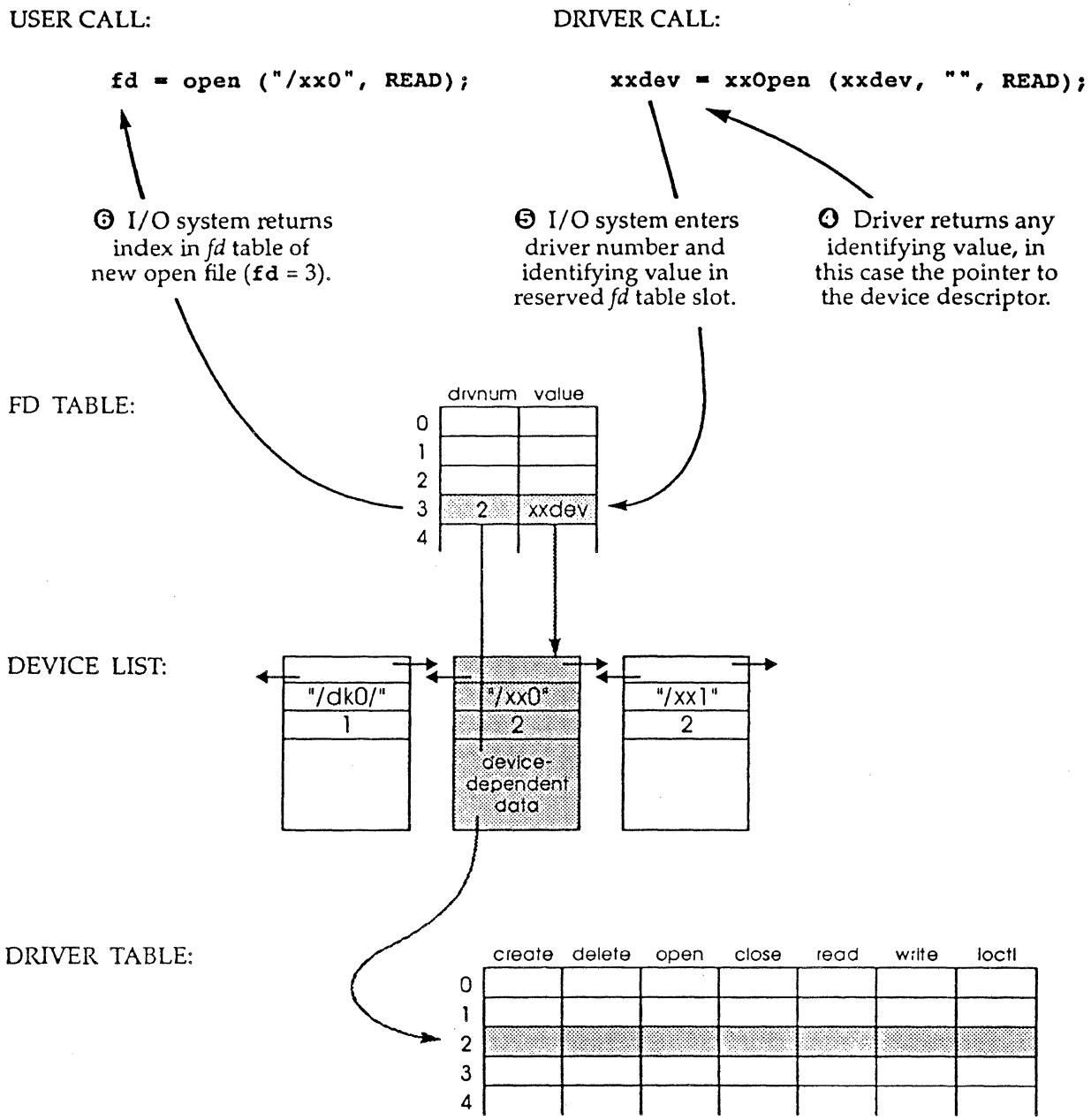


Figure 4-5. Example – Call to I/O Routine *open*( ) (*Part 2*)

device. In the case of non-block devices like this one, it is an error for the remainder to be anything *but* the null string.

❹ It executes *xxOpen*( ), which returns a value by which the opened file is identified. In this case, the value is the pointer to the device descriptor. This value is supplied to the driver in subsequent I/O calls that refer to the file being opened. If the driver returns just the device descriptor, the driver is not able to distinguish multiple files opened to the same device. In the case of non-block device drivers, this is appropriate.

❺ The I/O system then enters the driver number and the value returned by *xxOpen*( ) in the reserved slot in the *fd* table. Again, the value entered in the *fd* table is meaningful to the driver and is arbitrary as far as the I/O system is concerned.

❻ Finally, the I/O system returns the index of the slot in the *fd* table, in this case 3.

4.9.3.3 Example of Reading Data from the File

In Figure 4-6, the user calls *read*( ) to obtain input data from the file. The specified *fd* is the index into the *fd* table for this file. The I/O system uses the driver number contained in the *fd* table to locate the driver's read routine, *xxRead*( ). The I/O system calls *xxRead*( ), passing it the identifying value in the *fd* table that was returned by the driver's open routine, *xxOpen*( ). Again, in this case the value is just the pointer to the device descriptor. The driver's read routine then does whatever is necessary to read data from the device.

The process for user calls to *write*( ) and *ioctl*( ) follow the same procedure.

4.9.3.4 Example of Closing a File

You terminate the use of a file by calling *close*( ). As in the case of *read*( ), the I/O system uses the driver number contained in the *fd* table to locate the driver's close routine. In the example driver, no close routine has been specified; therefore, no driver routines are called. Instead, the I/O system marks the slot in the *fd* table as being available. Any subsequent references to that *fd* causes an error. However, subsequent calls to *open*( ) can reuse that slot.

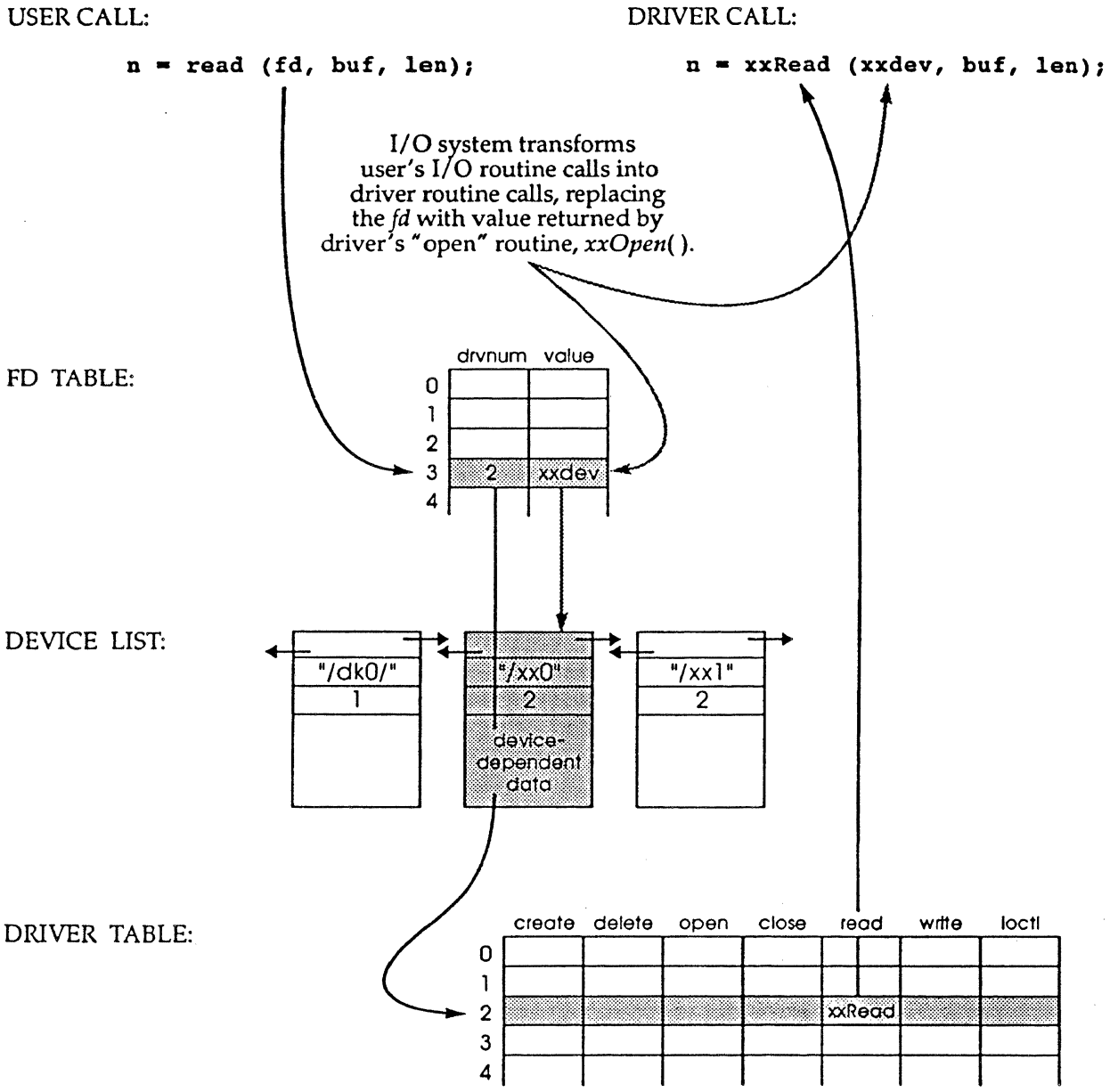


Figure 4-6. Example – Call to I/O Routine *read( )*

4.9.4 Block Devices

4.9.4.1 General Implementation

In Vx960, block devices have a different interface than other I/O devices. Rather than interacting with the I/O system, block device drivers interact with a file system. The file system, in turn, interacts with the I/O system. Figure 4-7 shows a layered model of I/O for both block and non-block devices. This arrangement allows the same block device driver to be used with different file systems and reduces the number of I/O functions that must be supported in the driver.

A device driver for a block device must provide a means for creating a logical block device structure, known as a BLK_DEV. The BLK_DEV structure describes the device in a generic fashion, specifying those common characteristics that must be known to a file system being used with the device. Fields within the structure specify various physical configuration variables for the device (e.g., block size, total number of blocks, etc.) as well as routines within the device driver which are to be used for manipulating the device (reading blocks, writing blocks, doing I/O control functions, resetting the device, and checking device status). The BLK_DEV structure also contains fields used by the driver to indicate certain conditions (e.g., a disk change) to the file system.

When the driver creates the block device, the device has no name or file system associated with it. These are assigned during the device initialization routine for the chosen file system (e.g., *dosFsDevInit*( ) or *rt11FsDevInit*( )).

The device driver for a block device is not installed in the I/O system driver table, as are non-block device drivers. Instead, each file system in the Vx960 system is installed in the table as a "driver." Each file system has one entry in the table, even though several different device drivers can have devices served by that file system.

After a device is initialized for use with a particular file system, all I/O operations for the device are routed through that file system. To perform specific device operations, the file system in turn calls the routines whose addresses were placed in the BLK_DEV structure.

The driver for a block device must provide the interface between the Vx960 and the device. Vx960 requires a specific set of functions; individual devices vary in terms of what additional functions must be provided. The user manual for the particular device used, as well as any other drivers for the device, is invaluable in creating the

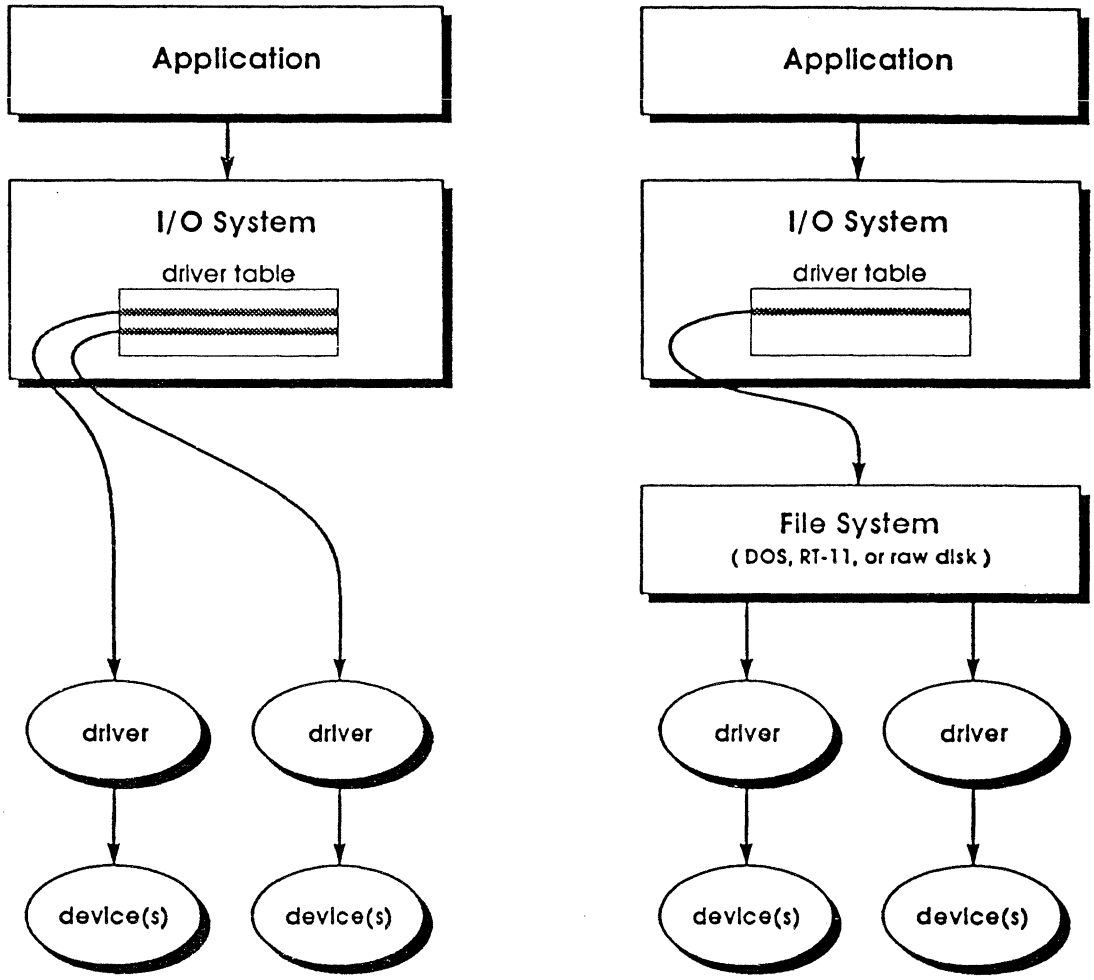


Figure 4-7. Implementation of Vx960 Device Drivers

Vx960 driver. The following sections describe the components necessary to build block device drivers that adhere to the standard interface for Vx960 file systems.

4.9.4.2 Driver Initialization Routine

The driver requires a general initialization routine. This routine should perform all operations which are done once only, as opposed to functions which must be performed for each device served by the driver. As a general guideline, the functions in the initialization routine affects the whole device controller, while later functions affect specific devices.

Common operations in block device driver initialization routines include:

- hardware initialization
- allocation and initialization of data structures
- semaphore creation
- interrupt vector initialization
- enabling interrupts.

The functions performed in the initialization routine are specific to the device (controller) being used; Vx960 has no requirements for a driver initialization routine.

Unlike non-block device drivers, the driver initialization routine does *not* call *iosDrvInstall*( ) to install the driver in the I/O system driver table. Instead, the file system used installs itself as a "driver". This file system routes calls to the actual driver via the routine addresses placed in the block device structure, BLK_DEV (see below).

4.9.4.3 Device Creation Routine

The driver must provide a routine that creates (defines) a logical disk device. This logical device can in fact be a portion of a larger physical device. If this is the case, it is the responsibility of the device driver to keep track of any block offset values or other means of identifying the physical area corresponding to the logical device. Vx960 file systems always use block numbers beginning with zero for the start of a device.

The device creation routine allocates a device descriptor structure which the driver uses to manage the device. The first item in this device descriptor must be a Vx960 block device structure (BLK_DEV). The BLK_DEV must appear first because its

address is passed by the file system during calls to the driver; by having the BLK_DEV be the first item, this address serves to identify the device descriptor.

The device creation routine must initialize the fields within the BLK_DEV structure. The BLK_DEV fields and their initialization values are shown in the following table:

Table 4-9. Fields in the BLK_DEV Structure

Field	Value
bd_blkRd	address of driver routine that reads blocks from the device.
bd_blkWrt	address of driver routine that writes blocks to the device.
bd_ioctl	address of driver routine that performs device I/O control.
bd_reset	address of driver routine that resets the device (NULL if none).
bd_statusChk	address of driver routine that checks disk status (NULL if none).
bd_removable	TRUE if device is removable (e.g., floppy disk); otherwise FALSE.
bd_nBlocks	total number of blocks on device.
bd_bytesPerBlk	number of bytes per block on device.
bd_blksPerTrack	number of blocks per track on device.
bd_nHeads	number of heads (surfaces).
bd_retry	number of times failed reads or writes should be retried.
bd_mode	device mode (write protect status); set to UPDATE.
bd_readyChanged	TRUE if device ready status has changed; should be initialized to TRUE to cause disk to be mounted.

The device creation routine should return the address of the BLK_DEV structure. This address is passed during the file system device initialization call to identify the device.

Unlike non-block device drivers, the device creation routine for a block device does not call *iosDevAdd( )* to install the device in the I/O system device table. This is done by the file system's device initialization routine.

4.9.4.4 Read-Blocks Routine

The driver must supply a routine that reads one or more blocks from the device. This routine must be defined as follows:

```
STATUS xxBlkRd (pDev, startBlk, numBlks, pBuf)
    DEVICE  *pDev;      /* pointer to device descriptor */
    int     startBlk;   /* starting block to read */
    int     numBlks;    /* number of blocks to read */
    char    *pBuf;      /* pointer to buffer to receive data*/
```

Note: In this and following examples, the routine names begin with "xx". These names are for illustration, and do not have to be used by your device driver. Vx960 references the routines by address only – the name can be anything.

The *pDev* parameter is a pointer to the driver's device descriptor structure, represented here by the symbolic name DEVICE. (The file system passes the BLK_DEV structure address; these are equivalent since the BLK_DEV is the first item in the device descriptor.) This identifies the device.

The *startBlk* parameter is the starting block number to be read from the device. The file system always uses block numbers beginning with zero for the start of the device. Any offset value used for this logical device must be added in by the driver.

The *numBlks* parameter specifies the number of blocks to be read. If the underlying device hardware does not support multiple-block reads, the driver routine must do the necessary looping to emulate this ability.

The *pBuf* parameter is the address where data read from the disk is to be copied.

The read-blocks routine should return OK if the transfer was completed, or ERROR if a problem occurred.

4.9.4.5 Write-Blocks Routine

The driver must supply a routine that writes one or more blocks to the device. The definition of this routine parallels that of the read-blocks routine:

```
STATUS xxBlkWrt (pDev, startBlk, numBlks, pBuf)
    DEVICE  *pDev;      /* pointer to device descriptor */
    int     startBlk;   /* starting block for write */
    int     numBlks;    /* number of blocks to write */
    char    *pBuf;      /* pointer to buffer of data to write */
```

The *pDev* parameter is a pointer to the driver's device descriptor structure.

The *startBlk* parameter is the starting block number to be written to the device.

The *numBlks* parameter specifies the number of blocks to be written. If the underlying device hardware does not support multiple-block writes, the driver routine must do the necessary looping to emulate this ability.

The *pBuf* parameter is the address of the data to be written to the disk.

The write-blocks routine should return OK if the transfer was completed, or ERROR if a problem occurred.

4.9.4.6 I/O Control Routine

The driver must provide a routine that can handle I/O control requests. In Vx960, most I/O operations beyond basic file handling are implemented through *ioctl*( ) functions. The majority of these are handled by the file system. However, if the file system does not recognize a request, it is passed to the driver's I/O control routine.

The driver's I/O control routine should be defined as follows:

```
STATUS xxIoctl (pDev, funcCode, arg)
    DEVICE  *pDev;      /* pointer to device descriptor */
    int     funcCode;   /* ioctl function code */
    int     arg;        /* function-specific argument */
```

The *pDev* parameter is a pointer to the driver's device descriptor structure.

The *funcCode* parameter is the *ioctl*( ) function that was requested. Standard Vx960 I/O control functions are defined in the include file ioLib.h. Other user-defined function code values can be used as required by your device driver. I/O control

functions supported by the DOS, RT-11, and raw file systems are summarized in **5. Local File Systems.**

The *arg* parameter is an argument specific to the particular *ioctl( )* function requested. Not all *ioctl( )* functions use this parameter.

The driver's I/O control routine takes the form of a multi-way switch statement, based on the function code. The driver's I/O control routine must supply a default case for function code requests it does not recognize. If such a request occurs, the I/O control routine should set *errno* to S_ioLib_UNKNOWN_REQUEST and return ERROR.

The driver's I/O control routine should return OK if the request was handled, otherwise ERROR.

4.9.4.7 Device Reset Routine

The driver should supply a routine that resets a specific device. This routine is called when a disk is first mounted by a Vx960 file system, and again during retry operations when a read or write fails.

The driver's device-reset routine is declared as follows:

```
STATUS xxReset (pDev)
    DEVICE  *pDev;
```

The *pDev* parameter is a pointer to the driver's device descriptor structure.

When called, this routine resets the device and controller. Other devices should not be reset, if it can be avoided. The routine should return OK if the device was reset, otherwise ERROR.

If no reset function is required for the device, this routine can be omitted. In this case, the bd_reset field in the BLK_DEV structure should be set to NULL by the device creation routine.

4.9.4.8 Status-Check Routine

The driver can provide a routine to check device status or perform other necessary functions that should precede open and create operations on the device. If present, this routine is called by the file system at the beginning of each *open*( ) or *creat*( ) on the device.

The status-check routine should be defined as follows:

```
STATUS xxStatusChk (pDev)
    DEVICE  *pDev;        /* pointer to device descriptor */
```

The *pDev* parameter is a pointer to the driver's device descriptor structure.

The routine should return OK if the open or create operation can continue. If it detects a problem with the device, the routine should set *errno* to some value indicating the problem, and return ERROR. If ERROR is returned, the file system does not continue operation.

A primary use of the status-check routine is to check for a disk change on devices that do not detect the change until after a new disk has been inserted. For such devices, the routine should determine whether a new disk has been inserted. If a new disk is present, the routine sets the bd_readyChanged field in the BLK_DEV structure to TRUE and then returns OK so that the open or create operation can continue. The new disk is mounted by the file system. (See **4.9.4.10 Change In Ready Status.**)

If no status-check routine is required by the device driver, the bd_statusChk field in the BLK_DEV structure is set to NULL by the device creation routine.

4.9.4.9 Write-Protected Media

The device driver can detect that the disk in place has been write-protected. If this is the case, the driver sets the bd_mode field in the BLK_DEV structure to READ. This can be done at any time (i.e., even after the device has been initialized for use with the file system). The file system checks this value and does not allow writes to the device until the bd_mode field is changed (to UPDATE) or the file system's mode change routine (e.g., *dosFsModeChange*( )) has been called to change the mode. (The bd_mode field is changed if the file system's mode change routine is used.)

4.9.4.10 Change In Ready Status

The driver should inform the file system whenever a change in the device's ready status is recognized. This change could be the changing of a floppy disk, or any other situation that would make it advisable for the file system to remount the disk.

To announce a change in ready status, the driver should set the bd_readyChanged field in the BLK_DEV structure to TRUE. This is recognized by the file system and the disk is remounted during the next I/O initiated on the disk. The file system sets the bd_readyChanged field to FALSE. The bd_readyChanged field should never be cleared by the device driver.

Setting bd_readyChanged to TRUE has the same effect as calling the file system's ready-change routine (e.g., *dosFsReadyChange*( )) or calling *ioctl*( ) with the FIODISKCHANGE function code.

A status-check routine (see above) can provide a convenient mechanism for asserting a ready-change, for devices that cannot detect a disk change until after the new disk has been inserted. If the status-check routine detects that a new disk has been inserted, it should set bd_readyChanged to TRUE. The status-check routine, if present, is called by the file system at the beginning of each open or create operation.

4.9.5 Driver Support Libraries

There are several subroutine libraries that can assist in writing device drivers. Using these libraries, drivers for most devices following standard protocols can be written with just a few pages of device-dependent code. See the manual entries on these libraries for details.

Table 4-10. Vx960 Driver Support Libraries

Library	Description
errnoLib(1)	Error status library
ftpLib(1)	ARPA File Transfer Protocol library
ioLib(1)	I/O interface library
iosLib(1)	I/O system library
intLib(1)	Interrupt support subroutine library
remLib(1)	Remote command library
rngLib(1)	Ring buffer subroutine library
tyLib(1)	Serial device driver subroutine library
wdLib(1)	Watchdog timer subroutine library

5

Local File Systems

Contents

Local File Systems

5.1 Introduction

Vx960 provides two local file systems appropriate for real-time use with block devices (disks): one is compatible with DOS file systems and the other with the RT-11 file system. The support libraries for these file systems are **dosFsLib**(1) and **rt11FsLib**(1). Vx960 also provides a simple "raw file system," which treats an entire disk much like a single large file. The support library for this "file system" is **rawFsLib**(1).

In Vx960, the file system is not tied to a specific type of block device or its driver. Vx960 block devices all use a standard interface so that file systems can be freely mixed with device drivers. Alternative, user-supplied file systems can be written and used by drivers in the same way, by following the same standard interfaces between the file system, the driver, and the Vx960 I/O system. Vx960 I/O architecture makes it possible to have multiple file systems, even of different types, at the same time in a single Vx960 system. The block device interface is discussed in detail in the previous chapter under **4.9.4 Block Devices.**

This chapter discusses the organization, configuration, and use of Vx960 file systems.

5.2 DOS-Compatible File System

The Vx960 DOS file system is media-compatible with versions of MS-DOS up to and including release 4.0. Vx960 DOS file system capabilities offer considerable flexibility appropriate to the varying demands of real-time applications. Major features include:

- A hierarchical arrangement of files and directories that allows efficient organization and permits an indefinite number of files to be created on a volume.
- A choice of contiguous or non-contiguous files on a per-file basis. Non-contiguous files result in more efficient use of available disk space while contiguous files offer enhanced performance.
- Compatibility with widely available storage and retrieval media. Disks created with Vx960 and MS-DOS PCs and other systems can be freely interchanged.

5.2.1 Disk Organization

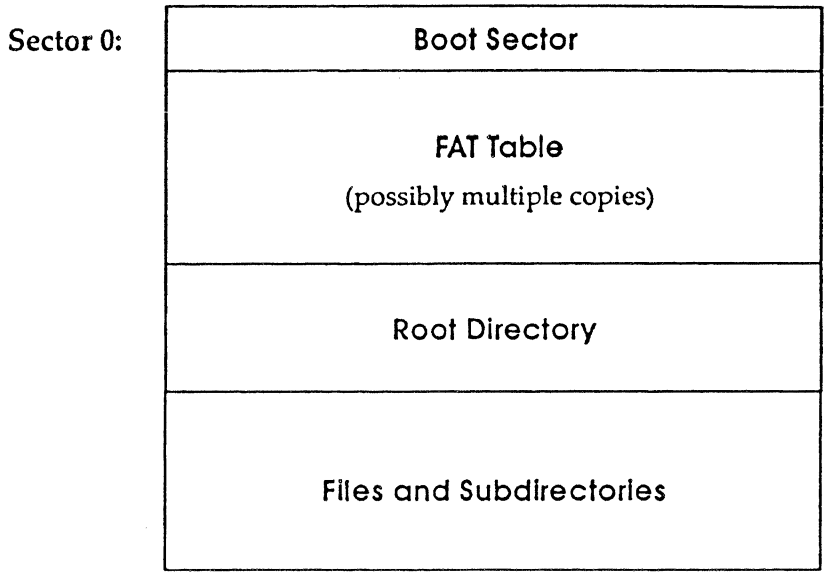
The Vx960 DOS-compatible file system provides the means for organizing disk data in a flexible manner. It maintains a hierarchical set of named directories, each containing files or other directories. Files can be appended, and as they expand, new disk space is allocated automatically. The allocation of disk space to files is not necessarily contiguous, which results in a minimum of wasted space. However, to enhance its real-time performance, the Vx960 DOS file system allows contiguous space to be pre-allocated to files individually, thereby minimizing seek operations.

The general organization of a DOS file system is shown in Figure 5-1, and the various elements are discussed in the following sections.

5.2.1.1 Clusters

A file in a DOS file system consists of one or more disk *clusters*. A cluster is a set of contiguous disk sectors[1]. For floppy disks, two sectors make up a cluster; for fixed

1. In this and subsequent sections covering the DOS file system, the term *sector* refers to the minimum addressable unit on a disk, since this is the term used by most MS-DOS documentation. In Vx960, the units are normally referred to as *blocks*, and a disk device is called a *block device*.



Note: If number of reserved sectors (dosvc_nResrvd) is greater than 1, the first FAT table copy will not immediately follow the boot sector.

Figure 5-1. DOS Disk Organization

disks, there may be more sectors per cluster. A cluster is the smallest amount of disk space the file system can allocate at a time. A large number of sectors per cluster allows a larger disk to be described in a fixed-size file allocation table (FAT– see below), but can result in wasted disk space.

5.2.1.2 Boot Sector

The first sector on a DOS disk is called the *boot sector*. This sector contains a variety of configuration data. Some of the data fields describe the physical properties of the disk (e.g., the total number of sectors), and other fields describe file system variables (e.g., the size of the root directory).

The boot sector information is written to the disk when it is initialized. The Vx960 DOS file system can use disks which have been initialized on another system (for example, using the FORMAT program on an MS-DOS PC), or Vx960 can initialize the disk itself, using the FIODISKINIT function with an *ioctl( )* call.

As the DOS standard has evolved, various fields have been added to the boot sector definition. Disks initialized under Vx960 use the boot sector fields defined by MS-DOS version 4.0.

5.2.1.3 File Allocation Table

Each DOS volume contains a File Allocation Table, commonly called the *FAT table*. The FAT table contains an entry for each cluster on the disk that can be allocated to a file or directory. When a cluster is unused (available for allocation) its entry is zero. If a cluster is allocated to a file, its entry is the cluster number of the next portion of the file. If a cluster is the last in a file, its entry is -1. Thus, the representation of a file (or directory) consists of a linked list of FAT table entries.

FAT tables use either 12 or 16 bits per entry. Disk volumes that contain up to 4085 clusters use 12-bit entries; disks with more than 4085 clusters use 16-bit entries. The entries themselves (particularly 12-bit entries) are encoded in a specific manner, done originally to take advantage of the Intel 8088 architecture. However, all FAT table handling is done by the Vx960 DOS file system; therefore, the encoding and decoding is of no concern to Vx960 applications.

A volume contains multiple copies of the FAT table. This redundancy allows data recovery in the event of a media error in the first FAT copy. The Vx960 DOS file system maintains multiple FAT copies; however, the copies are not automatically used in the event of an error.

The size of the FAT table and the number of FAT copies are determined by fields in the boot sector. For disks initialized using the Vx960 DOS file system, these parameters are specified during the *dosFsDevInit( )* call by setting fields in the volume configuration structure, DOS_VOL_CONFIG.

5.2.1.4 Root Directory

Each DOS volume contains a root directory. The root directory occupies a set of contiguous disk sectors following the FAT table copies. The disk area occupied by the root directory is not described by entries in the FAT table.

The root directory is of a fixed size; this size is specified by a field in the boot sector as the maximum allowed number of directory entries. For disks initialized using the Vx960 DOS file system, this size is specified during the *dosFsDevInit*( ) call, by setting a field in the volume configuration structure, DOS_VOL_CONFIG.

Since the root directory has a fixed size, attempts to add entries once the directory is full returns an error.

For information on the contents of the directory entry, see **5.2.2.11 DOS Directory Entries.**

5.2.1.5 Subdirectories

In addition to the root directory, DOS volumes may contain a hierarchy of subdirectories. Like the root directory, subdirectories contain entries for files and other subdirectories; however, in other ways they differ from the root directory and resemble files:

First, they are described by an entry in another directory, as is a file. The directory entry has a bit set in the file attribute byte to indicate that it describes a subdirectory. Because they are described by directory entries, subdirectories have names, unlike the root directory.

Second, subdirectories are composed of a set of disk clusters, linked by FAT table entries. Subdirectories may grow as entries are added to it and subdirectories are not necessarily made up of contiguous clusters. The root directory, unlike subdirectories, can be made up of any number of sectors, not necessarily equal to a whole number of clusters.

Third, subdirectories always contain two special entries. The "." entry refers to the subdirectory itself, while the ".." entry refers to the subdirectory's parent directory. The root directory does not contain these special entries.

5.2.1.6 Files

A file in the DOS file system is a set of clusters that are chained together via entries in the FAT table. A file is not necessarily made up of contiguous clusters; the various clusters may be located anywhere on the disk and in any order.

Each file has a descriptive entry in its parent directory. This entry contains the file's name, size, modification date and time, and a special field giving the file's attribute (e.g., a read-only file). It also contains the starting cluster number for the file; subsequent clusters are located using the FAT table.

5.2.1.7 Volume Label

A DOS disk may have a *volume label* associated with it. The volume label is a special entry in the root directory. Rather than containing the name of a file or subdirectory, the volume label entry contains a string used to identify the volume. This string can contain up to 11 characters. The volume label entry is identified by a special value of the file attribute byte in the directory entry.

The volume label can be added to a Vx960 DOS volume by using the *ioctl( )* call with the **FIOLABELSET** function. This will add a label entry to the volume's root directory if none exists, or it will change the label string in an existing volume label entry. The volume label entry takes up one of the fixed number of root directory entries; attempting to add a volume label entry when the root directory is full will result in an error.

The current volume label string for a volume can be obtained by using the *ioctl( )* function with the **FIOLABELGET** function. If the volume has no label, this call will return **ERROR** and will set *errno* to **S_dosFsLib_NO_LABEL**.

Disks initialized under Vx960 or under MS-DOS 4.0 (or later) also contain the volume label string within a boot sector field.

5.2.2 Using the DOS File System

5.2.2.1 Initializing the File System

Before you can perform any other operations, you must initialize the Vx960 DOS file system library, dosFsLib(1) by calling *dosFsInit( )*. This routine takes a single parameter, the maximum number of dosFs file descriptors that can be open at one time. This count is used to allocate a set of file descriptors which can be used as files, directories, or the file-system device.

Note: Vx960 includes optional features and device drivers that can be added or omitted from the system. These features and drivers are controlled by special symbols that can be defined in the configuration headers to cause conditional compilation in the usrConfig.c module. For a list of the optional elements and their control symbols, refer to Table 8-1.

The *dosFsInit*( ) routine also makes an entry for the DOS file system in the I/O system system driver table (via *iosDrvInstall*( )). This entry defines specific entry points for DOS file operations and is used for all devices which use the DOS file system. The driver number assigned to the DOS file systems is placed in a global variable *dosFsDrvNum*( ).

The *dosFsInit*( ) routine also makes an entry for the DOS file system in the I/O system driver table (via *iosDrvInstall*( )). This entry defines specific entry points for DOS file operations and is used for all devices which use the DOS file system. The driver number assigned to the DOS file system is placed in a global variable *dosFsDrvNum*( ).

If you prefer to initialize the DOS file system at a different time, do not define INCLUDE_DOSFS in configAll.h. *dosFsInit*( ) can then be called at any later time; however, it should not be called more than once.

5.2.2.2 Initializing a Device for Use with DOS

Once the DOS file system has been initialized, the next step is to create one or more devices. Devices are created by the device driver's device creation routine. The driver routine returns a pointer to a block device descriptor structure (BLK_DEV). The BLK_DEV structure describes the physical aspects of the device and specifies the routines in the device driver that a file system will use.

After its creation, the block device has neither a name nor a file system associated with it. To initialize a block device for use with the DOS file system, the already-created block device must be associated with dosFs and a name must be assigned to it. Do this with the *dosFsDevInit*( ) routine. Its parameters are the name to be used to identify the device, a pointer to the block device descriptor structure (BLK_DEV), and a pointer to the volume configuration structure DOS_VOL_CONFIG (see **5.2.2.3 Volume Configuration** below). For example:

```
DOS_VOL_DESC *pVolDesc;

pVolDesc = dosFsDevInit ("DEV1:", pBlkDev, &configStruct);
```

The *dosFsDevInit*( ) call performs the following tasks:

- assigns the specified name to the device and enters the device in the I/O system device table (via *iosDevAdd*( )).
- allocates and initializes the file system's volume descriptor for the device.
- returns a pointer to the volume descriptor. This pointer is used to identify the volume during certain file system calls.

Initializing the device for use with a DOS file system does not format the disk, nor does it initialize the DOS disk structures (root directory, FAT table, etc.). These are done using the *ioctl*( ) functions FIODISKFORMAT and FIODISKINIT, respectively.

5.2.2.3 Volume Configuration

The volume configuration structure, DOS_VOL_CONFIG, is used during the *dosFsDevInit*( ) call. This structure contains various DOS file system variables describing the layout of data on the disk. The majority of the fields in the structure correspond to those found in the boot sector.

The *dosFsConfigInit*( ) call provides a convenient way to initialize the DOS_VOL_CONFIG structure. It takes the configuration variables as parameters and fills in the structure. This is useful for initializing devices interactively from the Vx960 shell. The DOS_VOL_CONFIG structure must be allocated *before* *dosFsConfigInit*( ) is called.

DOS_VOL_CONFIG Fields

The table below lists the fields in the DOS_VOL_CONFIG structure:

Table 5-1. DOS_VOL_CONFIG Fields

Function	Meaning
dosvc_secPerClust	number of sectors per cluster
dosvc_nResrvd	number of reserved sectors that precedes the first FAT table copy; the minimum is 1 (the boot sector)
dosvc_nFats	number of FAT table copies
dosvc_maxRootEnts	maximum number of entries in root directory
dosvc_mediaByte	media descriptor byte
dosvc_secPerFat	number of sectors per FAT table copy
dosvc_nHidden	number of hidden sectors, normally 0
dosvc_changeNoWarn	TRUE = disk change is unannounced
dosvc_autoSync	TRUE = synchronize disk during I/O

All but the last two DOS_VOL_CONFIG fields in Table 5-1 describe standard DOS characteristics; the last two are specific to the Vx960 DOS file system. These two fields specify the action used to synchronize the disk buffers to the physical device:

dosvc_changeNoWarn: This field should be set to TRUE if the device is a disk that might be replaced without being unmounted or a change in ready status being declared. In this situation, it is necessary to recheck the disk regularly to determine if it has been changed. Since this causes significant overhead, it is desirable to provide a mechanism for synchronizing and unmounting a disk before it is removed, or at least announcing a change in ready status. If such a mechanism is in place, or if the disk is not removable, the dosvc_changeNoWarn field should be defined as FALSE. If dosvc_changeNoWarn is set to TRUE, auto-sync mode is automatically enabled (see below). For more information on the use of this field, see **5.2.2.15 Changing Disks.**

dosvc_autoSync: Setting this field to TRUE assures that the data in the disk's buffers will be written to the physical device as soon as possible after modification, rather than only when the file is closed. This may be desirable for critical data in situations where it is important that a maximum amount of data be stored safely on the phys-

ical media, e.g., to avoid loss in the event of a system crash. There is a significant performance penalty incurred when using auto-sync mode; its use should therefore be limited to special circumstances where data integrity is of greater concern than performance. Auto-sync mode is automatically enabled if dosvc_changeNoWarn is set to TRUE. For more information on auto-sync mode, see **5.2.2.15 Changing Disks**.

Calculating Configuration Values

The values for dosvc_secPerClust and dosvc_secPerFat must be calculated based on the particular device being used.

dosvc_secPerClust: This field specifies how many contiguous disk sectors make up a single cluster. Since a cluster is the smallest amount of disk space that can be allocated at a time, the size of a cluster determines how finely the disk allocation can be controlled. A large number of sectors per cluster causes more sectors to be allocated at a time and reduces the overall efficiency of disk space usage. For this reason, it is generally preferable to use the smallest possible number of sectors per cluster, although having less that 2 sectors per cluster is generally not necessary.

Since the maximum size of a FAT table entry is 16 bits, there is a maximum of 65536 (64K, or 0x10000) clusters which can be described. This is therefore the maximum number of clusters for a device. To determine the number of sectors per cluster, divide the total number of sectors on the disk (i.e., the bd_nBlocks field in the device's BLK_DEV structure) by 0x10000 (64K). The resulting value should then be rounded up to the next whole number. The final result is the number of sectors per cluster, and this value should be placed in the dosvc_secPerClust field in the DOS_VOL_CONFIG structure.

dosvc_secPerFat: This field specifies the number of sectors required on the disk for each copy of the FAT table. To calculate this value, you must first determine the total number of clusters on the disk. The total number of clusters is equal to the total number of sectors (bd_nBlocks in the BLK_DEV structure) divided by the number of sectors per cluster. As mentioned above, the maximum number of clusters on a disk is 64K.

The cluster count must then be multiplied by the size of each FAT entry: if the total number of clusters is 4085 or less, each FAT entry requires 1.5 bytes; if the number of clusters is greater than 4085, each FAT entry requires 2 bytes. The result of this multiplication is the total number of bytes required by each copy of the FAT table. This byte count is then divided by the size of each sector (i.e., the bd_bytesPerBlk field in the BLK_DEV structure) to determine the number of sectors required for each FAT

copy; if there is any remainder, add one (1) to the result. This final value should be placed in the dosvc_secPerFat field.

Assuming 512-byte sectors, the largest possible FAT table (with entries describing 64K clusters) occupies 256 sectors per copy, calculated as follows:

$$\frac{64\text{K entries} \times 2 \text{ bytes/entry}}{512 \text{ bytes/sector}} = 256 \text{ sectors}$$

Standard Disk Configurations

For floppy disks, there are a number of standard disk configurations used in DOS systems. In general, these are identified by the media descriptor byte value (at least for a given size of floppy disk), although some manufacturers have used duplicate values for different formats. Some widely used configurations are summarized in Table 5-2 below.

Fixed disks do not use such standard disk configurations, since they seldom are attached to a foreign system. In general, fixed disks use a media format byte of 0xF8.

Table 5-2. DOS Floppy Disk Configurations

Size	Capacity	Sides	Tracks	Sectors/Track	Bytes/Sector	secPerClust	nResrvd	nFats	maxRootEnts	mediaByte	secPerFat	nHidden
5.25"	160K	1	40	8	512	1	1	2	64	0xFE	1	0
5.25"	180K	1	40	9	512	1	1	2	64	0xFC	2	0
5.25"	320K	2	40	8	512	2	1	2	112	0xFF	1	0
5.25"	360K	2	40	9	512	2	1	2	112	0xFD	2	0
5.25"	1.2M	2	80	15	512	1	1	2	224	0xF9	7	0
3.5"	720K	2	80	9	512	2	1	2	112	0xF9	3	0
3.5"	1.44M	2	80	18	512	1	1	2	224	0xF0	9	0

5.2.2.4 Changes In Volume Configuration

As mentioned above, various disk configuration parameters are specified when the DOS file system device is first initialized using *dosFsDevInit( )*. This data is kept in the volume descriptor, DOS_VOL_DESC, for the device. However, it is possible for a disk with different parameter values to be placed in a drive after the device has already been initialized. For such a disk to be usable, you must modify the configuration data in the volume descriptor when a new disk is present.

When a disk is mounted, the boot sector information is read from the disk. This data is used to update the configuration data in the volume descriptor. This happens the first time the disk is accessed, and again after the volume has been unmounted (using *dosFsVolUnmount( )*) or a ready-change operation is performed. For more information, see 5.2.2.15 Changing Disks

This automatic re-initialization of the configuration data has an important implication. Since the volume descriptor data is used when initializing a disk (with FIODISKINIT) it initializes a disk with the configuration of the most recently mounted disk, regardless of the original specification during *dosFsDevInit( )*. Therefore, we recommend that you use FIODISKINIT after *dosFsDevInit( )* and before any disk has been mounted. (The device is opened in raw mode, the FIODISKINIT I/O control call is performed, and the device is closed.)

5.2.2.5 Using an Already Initialized Disk

If you are using a disk that has already been initialized with a DOS boot sector, FAT table, and root directory (for example, by using the PC-based FORMAT utility), it is not necessary to provide the volume configuration data during *dosFsDevInit( )*.

Since the values in the volume descriptor are reset when a volume is mounted, it is possible to omit the DOS configuration data (by specifying a NULL pointer instead of the address of a DOS_VOL_CONFIG structure during *dosFsDevInit( )*), provided it is absolutely certain that the first use of the volume will be with a properly formatted and initialized disk.

5.2.2.6 Mounting Volumes

A disk volume is mounted automatically during the first *open( )* or *creat( )* operation for a file or directory on the disk. (Certain *ioctl( )* calls also cause the disk to be mounted.) If a NULL pointer is specified instead of the address of a DOS_VOL_CON-

FIG structure during the *dosFsDevInit( )* call, the disk is mounted immediately to obtain the configuration values.

When a disk is mounted, the boot sector, FAT table, and directory data are read from the disk. The volume descriptor, DOS_VOL_DESC, is updated to reflect the configuration of the newly mounted disk.

Automatic mounting reoccurs on the first file access following *dosFsVolUnmount( )* or a ready-change operation (see **5.2.2.15 Changing Disks**) or periodically if the disk is defined during the *dosFsDevInit( )* call with the **dosvc_changeNoWarn** field set to TRUE. Automatic mounting does not occur when a disk is opened in raw mode (see **5.2.2.8 Opening the Whole Device (Raw Mode)**).

5.2.2.7 File I/O

Files on a DOS file system device are created, deleted, written, and read using the standard Vx960 I/O routines (*creat( )*, *delete( )*, *write( )*, and *read( )*). See **4.3 Basic I/O** in the previous chapter for more information.

5.2.2.8 Opening the Whole Device (Raw Mode)

It is possible to open an entire DOS volume, by specifying only the device name during the *open( )* or *creat( )* call. A file descriptor is returned, as when opening a regular file; however, operations on that file descriptor affect the entire device. Opening the entire volume in this manner is called *raw mode*.

The most common reason for opening the entire device is to obtain a file descriptor to perform an *ioctl( )* function that does not pertain to an individual file. An example is the FIONFREE function, which returns the number of available bytes on the volume. For many of these functions, however, the file descriptor can be any open file descriptor to the volume, even one for a specific file.

When a disk is initialized with DOS data structures (boot sector, empty root directory, FAT table), the device should be opened in raw mode. The *ioctl( )* function FIODISKINIT performs the initialization.

A disk can be read or written in raw mode. In this case, the entire disk data area (i.e., the disk portion following the boot sector, root directory, and FAT table) is treated much like a single large file. No directory entry is made to describe any data written

using raw mode. For low-level I/O to the entire device, including the area used by DOS data structures, see **rawFsLib(1)**.

5.2.2.9 Creating Subdirectories

Subdirectories can be created in any DOS directory at any time, except in the root directory if it has reached its maximum entry count. Subdirectories may be created in either of two ways:

- Using *ioctl( )* with the FIOMKDIR function: The name of the directory to be created is passed as a parameter to *ioctl( )*. The file descriptor used for the *ioctl( )* call may have been acquired through opening the entire volume (raw mode), a regular file, or another directory on the volume.
- Using *open( )*: To create a directory, the O_CREAT option must be set in the "flags" parameter to open, and the FSTAT_DIR option must be set in the "mode" parameter. The *open( )* call returns a file descriptor which describes the new directory; this file descriptor can be used for reading only and should be closed when no longer needed. This second method of directory creation is the one used by the Vx960 command *mkdir( )* in **usrLib**.

When creating a directory using either method, the name of the new directory must be specified. This name can be either a full path name or a path name relative to the current working directory.

5.2.2.10 Removing Subdirectories

There are also two methods to remove a directory. A directory must be empty (except for the "." and ".." entries) to be deleted. The root directory can never be deleted.

The first method of removing a directory is to use an *ioctl( )* call with the FIORMDIR function code. The name of the directory is given as a parameter to *ioctl( )*. Again, the file descriptor used can refer to any file or directory on the volume, or to the entire volume itself.

The second way to remove a directory is to use the *delete( )* function, specifying the name of the directory. This is the method used by the *rmdir( )* command in **usrLib**.

5.2.2.11 DOS Directory Entries

Each DOS directory contains a set of entries describing files and subdirectories. Each entry contains the following information about a file or subdirectory:

file name — an 8-byte string (padded with spaces, if necessary) specifying the base name of the file.

file extension — a 3-byte string (space-padded) specifying the optional extension portion of the file name. Subdirectory names cannot use the extension portion.

file attribute — a one-byte field specifying file characteristics. (See **5.2.2.13 File Attributes** below.)

time — the encoded creation or modification time for the file.

date — the encoded creation or modification date for the file.

cluster number — the number of the starting cluster within the file. Subsequent clusters are found by searching the FAT table.

file size — the size of the file, in bytes. This field is always 0 for entries describing subdirectories.

5.2.2.12 Reading Directory Entries

Directories on Vx960 DOS volumes can be searched using the *opendir( )*, *readdir( )*, *rewinddir( )*, and *closedir( )* routines. These calls can be used to determine the names of files and subdirectories.

To obtain more detailed information about a specific file, use the *fstat( )* or *stat( )* function. Along with standard file information, the structure used by these routines also returns the file attribute byte from a directory entry. For more information, see **dirLib**(1).

5.2.2.13 File Attributes

The file attribute byte in a DOS directory entry consists of a set of flag bits, each indicating a particular file characteristic. The characteristics described by the file attribute byte are:

Table 5-3. Flags in the File Attribute Byte		
Hex value	**Vx960 Flag Name**	**Meaning**
0x01	DOS_ATTR_RDONLY	read-only file
0x02	DOS_ATTR_HIDDEN	hidden file
0x04	DOS_ATTR_SYSTEM	system file
0x08	DOS_ATTR_VOL_LABEL	volume label (not a file)
0x10	DOS_ATTR_DIRECTORY	subdirectory
0x20	DOS_ATTR_ARCHIVE	file is subject to archiving

The read-only flag, DOS_ATTR_RDONLY, is checked when a file is opened for WRITE or UPDATE. If the flag is set, the open fails.

The hidden file and system file flags, DOS_ATTR_HIDDEN and DOS_ATTR_SYSTEM, are ignored by dosFsLib. If present, they are kept intact, but no special handling results (for example, these entries are reported when searching directories).

The volume label flag, DOS_ATTR_VOL_LABEL, indicates that this directory entry contains the DOS volume label for this disk. There can be only one volume label entry per volume, and it must be in the root directory. A volume does not require a label. The volume label entry is not reported when reading the contents of a directory (using *readdir*( )). It can only be determined using the *ioctl*( ) function FIOLABELGET. The volume label can be set (or reset) to any string of 11 or fewer characters, using the *ioctl*( ) function FIOLABELSET. Any file descriptor open to the volume can be used during these *ioctl*( ) calls.

The directory flag, DOS_ATTR_DIRECTORY, indicates that this entry is not a regular file but a subdirectory.

The archive flag, DOS_ATTR_ARCHIVE, is set when a file is created or modified. It is never cleared by Vx960. This flag is intended for use by other DOS programs that search a volume for modified files and selectively archive them. Such a program should also clear the archive flag.

All the flags in the attribute byte, except the directory and volume label flags, can be set or cleared using the *ioctl( )* function FIOATTRIBSET. This function is called after opening the specific file whose attributes are to be changed. The attribute byte value specified in the FIOATTRIBSET call is copied directly; to preserve existing flag settings, the current attributes should first be determined using *stat( )* or *fstat( )* and then changed using bitwise *and* or *or* operations. For example, to make a file read-only, while leaving other attributes intact:

```
fd = open ("file", READ, 0);        /* open file              */
fstat (fd, &statStruct);            /* get dir entry data     */

ioctl (fd, FIOATTRIBSET, (statStruct.st_attrib | DOS_ATTR_RDONLY));
                                    /* set read-only flag     */
close (fd);                         /* close file             */
```

5.2.2.14 File Date and Time

Directory entries contain a time and date for each file or directory. This time is set when the file is created, and it is updated at the close of a file that has been modified. Entries describing subdirectories are not updated – they always contain the creation date and time for the subdirectory.

dosFsLib maintains the date and time in an internal structure. While there is currently no mechanism for automatically advancing the date or time, two different methods for setting the date and time are provided.

The first method involves using two routines, *dosFsDateSet( )* and *dosFsTimeSet( )*. The following examples illustrate their use:

```
dosFsDateSet (1990, 12, 25);        /* set date to Dec-25-1990 */

dosFsTimeSet (14, 30, 22);          /* set time to 14:30:22    */
```

Call these routines periodically to update the time and date values.

The second method requires a user-supplied hook routine. If a time and date hook routine is installed using *dosFsDateTimeInstall( )*, the routine will be called whenever **dosFsLib** requires the current date and time. This facility is provided to take advantage of hardware time-of-day clocks which can be read to obtain the current time. It can also be used with other user-supplied application programs that maintain actual time and date.

The date/time hook routine should be defined as follows (the name *dateTimeHook* is an example, the actual routine name can be anything):

```
VOID dateTimeHook (pDateTime)
    DOS_DATE_TIME  *pDateTime;   /* ptr to DOS date & time struct */
```

On entry to the hook routine, the DOS_DATE_TIME structure contains the last time and date that was set in dosFsLib. The hook routine should then fill the structure with the correct values for the current time and date. Unchanged fields in the structure retain their previous values.

The DOS specification only provides for 2-second granularity for file time stamps. If the number of seconds in the time specified during *dosFsTimeSet( )* or the date/time hook routine is odd, it is rounded down to the next even number.

The date and time used by dosFsLib is initially Jan-01-1980, 00:00:00.

5.2.2.15 Changing Disks

To increase performance, the Vx960 DOS file system keeps copies of directory entries and the file allocation table (FAT) for each volume in memory. While this greatly speeds up access to files, it requires that dosFsLib be notified when removable disks are changed (e.g., floppies are swapped). Two different notification methods are provided and discussed below: (1) *dosFsVolUnmount( )* and (2) the ready-change mechanism.

Unmounting Volumes

The preferred method of announcing a disk change is to call *dosFsVolUnmount( )* before removing the disk. This call flushes all modified data structures to disk if possible (see description of disk synchronization, below) and also marks any open file descriptors as obsolete. During the next I/O operation, the disk is remounted. The *ioctl( )* call can also be used to initiate *dosFsVolUnmount( )*, by specifying the FIOUNMOUNT function code. Any open file descriptor to the device can be used in the *ioctl( )* call.

Subsequent attempts to use obsolete file descriptors for further I/O operations will return an S_dosFsLib_FD_OBSOLETE error. To free such file descriptors, use the *close( )* call, as usual. This frees the descriptor, but still returns S_dosFsLib_FD_OBSOLETE. File descriptors acquired when opening the entire volume (raw mode) are not marked as obsolete during *dosFsVolUnmount( )* and can still be used.

Interrupt handlers must not call *dosFsVolUnmount( )* directly, since it is possible for the call to pend while the device becomes available. The interrupt handler can instead give a semaphore that prepares a task to unmount the volume. (*dosFsReadyChange( )* can be called directly from interrupt handlers.)

When *dosFsVolUnmount( )* is called, it attempts to write buffered data out to the disk. It's use is inappropriate for situations where the disk-change notification does not occur until a new disk has been inserted — the old buffered data would be written to the new disk. In these circumstances, use *dosFsReadyChange( )* (see below).

If *dosFsVolUnmount( )* is called after the disk is physically removed, the data flushing portion of its operation fails. However, the file descriptors is still marked as obsolete, and the disk is marked as requiring remounting. An error is *not* returned by *dosFsVolUnmount( )* in this situation. To avoid lost data in such a situation, synchronize the disk before you remove it (see **Synchronizing Volumes** below).

Announcing Disk Changes with Ready-Change

The second method of informing **dosFsLib** that a disk change is taking place is via the *ready-change* mechanism. A change in the disk's ready status is interpreted by **dosFsLib** to indicate that the disk should be remounted during the next I/O operation.

There are three ways to announce a ready-change:

- Call the *dosFsReadyChange( )* routine directly.
- Use the *ioctl( )* call with the FIODISKCHANGE function code.
- The device driver can set the **bd_readyChanged** field in the BLK_DEV structure to TRUE; this has the same effect as notifying **dosFsLib** directly.

The ready-change mechanism does not provide the ability to flush data structures to the disk. It merely marks the volume as needing remounting. As a result, you can lose buffered data (data written to files, directory entries, or FAT table changes). You can avoid this lost data by synchronizing the disk before asserting ready-change (see **Synchronizing Volumes** below). The combination of synchronizing and asserting ready-change provides all the functionality of *dosFsVolUnmount( )* except for marking file descriptors as obsolete.

Since it does not attempt to flush data or perform other operations that could cause delay, ready-change can be used in interrupt handlers.

The block device driver status-check routine (identified by the bd_statusChk field in the BLK_DEV structure) can be useful for asserting ready-change for devices which only detect a disk change after the new disk has been inserted. This routine is called at the beginning of each *open( )* or *creat( )* operation, before the file system checks for ready-change. See **4.9.4 Block Devices.**

Disks with No Change Notification

If it is not possible for *dosFsVolUnmount( )* to be called or a ready-change to be announced each time the disk is changed, the device must be identified when it is initialized for use with the file system. You can do this by setting the dosvc_changeNoWarn field of the DOS_VOL_CONFIG structure to TRUE when calling *dosFsDevInit( )* (see **5.2.2.3 Volume Configuration**).

This configuration option results in a significant performance disadvantage, since the disk configuration data must be read in from the physical disk (in case it has been removed and a new one inserted). In addition, setting dosvc_changeNoWarn to TRUE also enables auto-sync mode (see below). All that is required for disk change notification is that either the *dosFsVolUnmount( )* call or ready-change be issued each time the disk is changed. It is not necessary that it be called from the device driver or an interrupt handler. For example, if your application provided a user- interface through which an operator could enter a command resulting in a *dosFsVolUnmount( )* call before removing the disk, that would be sufficient, and dosvc_changeNoWarn should be defined as FALSE. It is important, however, that the operator follow such a procedure strictly.

Synchronizing Volumes

When a disk is *synchronized,* all buffered data that has been modified is written to the physical device so that the disk is "up-to-date." This includes data written to files, updated directory information, and the FAT table.

To avoid loss of data, synchronize a disk before it is removed. It may or may not be necessary to explicitly synchronize a disk, depending on when (or if) the *dosFsVolUnmount( )* call is issued.

When *dosFsVolUnmount( )* is called, an attempt is made to synchronize the device before unmounting. If the disk is still present and writable at the time of the call, synchronization takes place; there is no need to synchronize the disk independently.

However, if the *dosFsVolUnmount( )* call is made after a disk has been removed, it is too late to synchronize – *dosFsVolUnmount( )* discards the buffered data. There-

fore, make a separate *ioctl( )* call specifying the FIOSYNC function before the disk is removed. (This could be done in response to an operator command.) The file descriptor used during the *ioctl( )* call should have been obtained by opening the whole volume (raw mode).

Auto-Sync Mode

dosFsLib provides a modified mode of synchronization called *auto-sync*. When this option is enabled, data for modified directories and the FAT table is written to the physical device as soon as these structures are altered. (Normally, such changes are not written out until the involved file is closed.)

Enable auto-sync mode by setting dosvc_autoSync in the DOS_VOL_CONFIG structure to TRUE when calling *dosFsDevInit( )*. It is automatically enabled if the volume does not have disk change notification (i.e., dosvc_changeNoWarn is TRUE during *dosFsDevInit( )*).

Auto-sync results in a performance penalty, but it provides the highest level of data security, since it minimizes the amount of time when directory and FAT data are not up-to-date on the disk. Auto-sync may be desirable for applications where data integrity in the event of a system crash is a greater concern than simple disk I/O performance.

5.2.2.16 Contiguous File Support

dosFsLib provides for efficient handling of *contiguous files*. A contiguous file is made up of a series of consecutive disk sectors. This capability includes both the allocation of contiguous space to a specified file (or directory) and optimized access to such a file.

To allocate a contiguous area to a file, the file is first created in the normal fashion, using *open( )* or *creat( )*. The file descriptor returned during the creation of the file is then used to make an the *ioctl( )* call, specifying the FIOCONTIG function. The parameter to the FIOCONTIG function is the size of the requested contiguous area, in bytes. The FAT table is searched for a suitable section of the disk, and if found, it is assigned to the file. (If there is no contiguous area on the volume large enough to satisfy the request, an error is returned.) The file can then be closed or used for further I/O operations. For example, the following creates a file and allocates 0x10000 contiguous bytes on disk to it:

```
fd = creat ("file", UPDATE, 0);              /* open file              */
status = ioctl (fd, FIOCONTIG, 0x10000);     /* get contiguous area    */
if (status != OK)
    /* do error handling */
close (fd);                                  /* close file             */
```

It is important that the file descriptor used for the *ioctl( )* call is the only descriptor open to the file. Furthermore, since a file could be assigned a different area of the disk than was originally allocated, the *ioctl( )* (FIOCONTIG) operation should take place before any data is written to the file.

After the contiguous space has been allocated to a file, the file's size (kept in its directory entry) remains unchanged. The size value is increased only as space is actually used by writing to the file.[2]

Subdirectories can also be allocated a contiguous disk area in the same manner. If the directory was created using the *ioctl( )* function FIOMKDIR, it must be explicitly opened to obtain a file descriptor to it; if the directory was created using options to *open( )*, the returned file descriptor from that call can be used. A directory should be empty (except for the "." and ".." entries) before it has contiguous space allocated to it.

When any file is opened, it is checked for contiguity. If a file is recognized as contiguous, more efficient techniques for locating specific sections of the file are used, rather than following cluster chains in the FAT table as must be done for fragmented files. This enhanced handling of contiguous files takes place regardless of whether the space was actually allocated using FIOCONTIG.

5.2.2.17 I/O Control Functions Supported by dosFsLib

The Vx960 DOS file system supports the following *ioctl( )* functions. The functions listed are defined in the header **ioLib.h**. For more information, see the manual entry for **dosFsLib**(1) and the manual entry for *ioctl( )* in **ioLib**(1).

2. If you examine Vx960 DOS disks by using a PC-based disk debugging utility, errors may be reported for files which have had additional space allocated to them via FIOCONTIG. This is due to mismatches between the file size and the amount of allocated space. However, there should be no difficulty in using such a disk for normal PC applications.

Table 5-4. I/O Control Functions Supported by dosFsLib

Function	Meaning
FIODISKFORMAT	Format the disk (device driver function).
FIODISKINIT	Initialize a DOS file system on a disk volume.
FIODISKCHANGE	Announce a media change.
FIOUNMOUNT	Unmount a disk volume.
FIOGETNAME	Get the file name of the *fd*.
FIORENAME	Rename a file.
FIOSEEK	Set the current byte offset in a file.
FIOWHERE	Return the current byte position in a file.
FIOFLUSH	Flush the file output buffer.
FIOSYNC	Same as FIOFLUSH.
FIONREAD	Get the number of unread bytes in a file.
FIONFREE	Get the number of free bytes on the volume.
FIOMKDIR	Create a new directory.
FIORMDIR	Remove a directory.
FIOLABELGET	Get the volume label.
FIOLABELSET	Set the volume label.
FIOATTRIBSET	Set the file attribute byte in the DOS directory entry.
FIOCONTIG	Allocate contiguous disk space for a file or directory.
FIOREADDIR	Read the next directory entry.
FIOFSTATGET	Get file status information (directory entry data).

5.3 RT-11 File System

5.3.1 Disk Organization

The RT-11 file system employs a very simple disk organization. Although this simplicity results in some loss of flexibility, RT-11 is suitable for a variety of real-time applications.

The RT-11 file system maintains only *contiguous* files. A contiguous file consists of a series of disk sectors that are consecutive. Contiguous files are well-suited to real-time applications, because little time is spent locating specific portions of a file. The disadvantage of using contiguous files exclusively is that a disk can gradually become fragmented, reducing the efficiency of the disk space allocation.

The RT-11 disk format uses a single directory to describe all files on the disk. The size of this directory is limited to a fixed number of directory entries. Along with regular files, unused areas of the disk are also described by special directory entries. These special entries are used to keep track of individual sections of free space on the disk.

5.3.2 Using the RT-11 File System

5.3.2.1 Initializing the File System

Before you can perform any other operations, you must initialize the Vx960 RT-11 file system library, rt11FsLib(1), by calling *rt11FsInit*( ). This routine takes a single parameter, the maximum number of file descriptors which can be open at one time. This count is used to allocate a set of descriptors which can be used as files or **rt11Fs** devices.

The *rt11FsInit*( ) routine also makes an entry for the RT-11 file system in the I/O system system driver table (via *iosDrvInstall*( )). This entry defines specific entry points for RT-11 file operations and is used for all devices which use the RT-11 file system. The drive number assigned to the RT-11 file systems is placed in a global variable *rt11FsDrvNum*( ).

The *rt11FsInit*( ) routine is called by the *usrRoot*( ) task after starting the Vx960 system. To use this initialization, make sure the symbol INCLUDE_RT11FS is defined in the configuration file configAll.h, and set NUM_RT11FS_FILES to the desired maximum open file count.

If you prefer to initialize the RT-11 file system at a different time, do not define INCLUDE_RT11FS in the configuration file. *rt11FsInit*( ) can then be called at a later time; however, it should not be called more than once.

5.3.2.2 Initializing a Device for Use with RT-11

Once the RT-11 file system has been initialized, the next step is to create one or more devices. Devices are created by the device driver's device creation routine. The driver routine returns a pointer to a block device descriptor structure (BLK_DEV). The BLK_DEV structure describes the physical aspects of the device and specifies the routines in the device driver that a file system will use.

After its creation, the block device has neither a name nor a file system associated with it. To initialize a block device for use with RT-11, the already-created block device must be associated with rt11Fs and a name must be assigned to it. This is done with *rt11FsDevInit*( ). Its parameters are:

- the name to be used to identify the device
- a pointer to the BLK_DEV structure
- a boolean value indicating whether the disk uses standard RT-11 skew and interleave
- the number of entries to be used in the disk directory (in some cases, the actual number is greater than the number specified)
- a boolean value indicating whether this disk is subject to being changed without the file system being notified.

For example:

```
RT_VOL_DESC  *pVolDesc;

pVolDesc = rt11FsDevInit ("DEV1:", pBlkDev, rtFmt, nEntries, changeNoWarn);
```

The *rt11FsDevInit*( ) call assigns the specified name to the device and enters the device in the I/O system device table (via *iosDevAdd*( )). It allocates and initializes the file system's volume descriptor for the device. A pointer to the volume descrip-

tor is returned to the caller; this pointer is used to identify the volume during some file system calls.

Initializing the device for use with the RT-11 file system does not format the disk, nor does it initialize the RT-11 disk directory. These are done using *ioctl( )* with the functions FIODISKFORMAT and FIODISKINIT, respectively.

5.3.2.3 Mounting Volumes

A disk volume is mounted automatically, during the first *open( )* or *creat( )* for a file or directory on the disk. (Certain *ioctl( )* calls also cause the disk to be mounted.) When a disk is mounted, the directory data is read from the disk.

Automatic mounting reoccurs on the first file access following a ready-change operation (see **Changing Disks**) or periodically if the disk is defined during the *rt11FsDevInit( )* call with the changeNoWarn parameter set to TRUE. Automatic mounting does not occur when a disk is opened in raw mode. For more information, see **5.3.2.5 Opening the Whole Device (Raw Mode)**.

5.3.2.4 File I/O

Files on an RT-11 file system device are created, deleted, written, and read using the standard Vx960 I/O routines (*creat( )*, *delete( )*, *write( )*, and *read( )*). For more information, see **4.3 Basic I/O**.

5.3.2.5 Opening the Whole Device (Raw Mode)

You can open an entire RT-11 volume by specifying only the device name during the *open( )* or *creat( )* call. A file descriptor is returned, as when opening a regular file; however, operations on that file descriptor affect the entire device. Opening the entire volume in this manner is called *raw mode*.

The most common reason for opening the entire device is to obtain a file descriptor to perform an *ioctl( )* function that does not pertain to an individual file. An example is the FIOSQUEEZE function, which combines fragmented free space across the entire volume.

When a disk is initialized with an RT-11 directory, the device should be opened in raw mode. The *ioctl( )* function FIODISKINIT performs the initialization.

A disk can be read or written in raw mode. In this case, the entire disk area is treated much like a single large file. No directory entry is made to describe any data written using raw mode, and care must be taken to avoid overwriting the regular RT-11 directory at the beginning of the disk. This type of I/O is also provided by **rawFsLib**(1).

5.3.2.6 Reclaiming Fragmented Free Disk Space

The contiguous file allocation scheme used by the RT-11 file system can gradually result in disk fragmentation. In this situation, the available free space on the disk is scattered in a number of small chunks. This reduces the ability of the system to create new files.

To correct this condition, **rt11FsLib** includes a special utility, *rt11FsSqueeze*( ). This routine moves files so that the free space is combined at the end of the disk.

When *rt11FsSqueeze*( ) is called, it is absolutely critical that there be no open files on the device. This routine can take time to execute, particularly with large disks.

5.3.2.7 Changing Disks

To increase performance, the Vx960 RT-11 file system keeps copies of directory entries for each volume in memory. While this speeds up access to files, it requires that **rt11FsLib** be notified when removable disks are changed (e.g., floppies are swapped). This notification is provided by the ready-change mechanism.

- **Announcing Disk Changes with Ready-Change**

A change in the disk's ready status is interpreted by rt11FsLib to indicate that the disk should be remounted during the next I/O operation. There are three ways to announce a ready-change:

- by calling *rt11FsReadyChange*( ) directly
- by calling *ioctl*( ) with FIODISKCHANGE
- by having the device driver set the bd_readyChanged field in the BLK_DEV structure to TRUE; this has the same effect as notifying rt11FsLib directly.

The ready-change announcement does not cause buffered data to be flushed to the disk. It merely marks the volume as needing remounting. As a result, data written

to files or directory-entry changes can be lost. To avoid this loss of data, close all files on the volume before you change the disk.

Since it does not attempt to flush data or perform other operations that could cause delay, ready-change can be used in interrupt handlers.

The block device driver status-check routine (identified by the bd_statusChk field in the BLK_DEV structure) can be useful for asserting ready-change for devices which only detect a disk change after the new disk has been inserted. This routine is called at the beginning of each *open( )* or *creat( )* operation, before the file system checks for ready-change.

- Disks with No Change Notification

If a ready-change cannot be announced each time the disk is changed, the device must be identified when it is initialized for use with the file system. This is done by setting the changeNoWarn parameter to TRUE when calling *rt11FsDevInit( )*.

When this parameter is defined as TRUE, the disk is regularly re-read to obtain the current directory information (in case the disk has been removed and a new one inserted). As a result, this option causes a significant loss in performance.

5.3.2.8 I/O Control Functions Supported by rt11FsLib

The RT-11 file system supports the *ioctl( )* functions as shown in Table 5-5. The functions listed are defined in the header ioLib.h. For more information, see the manual entry for rt11FsLib(1) and the manual entry for *ioctl( )* in ioLib(1).

Table 5-5. I/O Control Functions Supported by rt11FsLib

Function	Meaning
FIODISKFORMAT	Format the disk.
FIODISKINIT	Initialize an RT-11 file system on a disk volume.
FIODISKCHANGE	Announce a media change.
FIOGETNAME	Get the file name of the *fd*.
FIORENAME	Rename a file.
FIOSEEK	Reset the current byte offset in a file.
FIOWHERE	Return the current byte position in a file.
FIOFLUSH	Flush the file output buffer.
FIONREAD	Get the number of unread bytes in a file.
FIOSQUEEZE	Coalesce fragmented free space on an RT-11 volume.
FIODIRENTRY	Get information about specified device directory entries.
FIOREADDIR	Read the next directory entry.
FIOFSTATGET	Get file status information (directory entry data).

5.4 Raw File System

Vx960 provides a minimal "file system" for use in systems which require only the most basic disk I/O functions. The "raw file system," implemented in **rawFsLib**, treats the entire disk volume much like a single large file. Although the Vx960 DOS and RT-11 file systems do provide this ability to varying degrees, the raw file system offers advantages in size and performance if more complex functions are not required.

5.4.1 Disk Organization

As mentioned above, **rawFs** imposes no organization of the data on the disk.

The raw file system does not maintain any directory information; therefore, no division of the disk area into specific files exists, and no file names are used. All *open( )* operations on **rawFs** devices specify only the device name; no additional file names are allowed.

The entire disk area is available to any file descriptor which is opened for the device. All read and write operations on the disk use a byte-offset relative to the start of the first block on the disk.

5.4.2 Using the Raw File System

5.4.2.1 Initializing the Raw File System

Before you can perform any other operations, you must initialize the Vx960 raw file system library, **rawFsLib**(1) by calling *rawFsInit( )*. This routine takes a single parameter, the maximum number of file descriptors which can be open at one time. This count is used to allocate a set of descriptors which can be used as **rawFs** devices.

The *rawFsInit( )* routine makes an entry for the raw disk file system in the I/O system driver table (via *iosDrvInstall( )*). This entry defines specific entry points for raw disk file operations and is used for all devices which use the raw disk file system. The drive number assigned to the raw disk file systems is placed in a global variable *rawFsDrvNum( )*.

The *rawFsInit( )* routine is called by the *usrRoot( )* task after starting the Vx960 system. To use this initialization, the symbol INCLUDE_RAWFS should be defined in **configAll.h**, and NUM_RAWFS_FILES should be set to the desired maximum open file descriptor count.

If you prefer to initialize the raw file system at a different time, do not define INCLUDE_RAWFS in **configAll.h**. You can call *rawFsInit( )* at a later time; however, do not call it more than once.

5.4.2.2 Initializing a Device for Use with the Raw File System

Once the raw file system has been initialized, the next step is to create one or more devices. Devices are created by the device driver's device creation routine. The driver routine returns a pointer to a block device descriptor structure (BLK_DEV). The BLK_DEV structure describes the physical aspects of the device and specifies the routines in the device driver that a file system will use.

Immediately after its creation, the block device has neither a name nor a file system associated with it. To initialize a block device for use with the raw file system, the already-created block device must be associated with rawFs and a name must be assigned to it. This is done with the *rawFsDevInit( )* routine. Its parameters are the name to be used to identify the device and a pointer to the block device descriptor structure (BLK_DEV):

```
RAW_VOL_DESC  *pVolDesc;

pVolDesc = rawFsDevInit ("DEV1:", pBlkDev);
```

The *rawFsDevInit( )* call assigns the specified name to the device and enters the device in the I/O system device table (via *iosDevAdd( )*). It also allocates and initializes the file system's volume descriptor for the device. A pointer to the volume descriptor is returned to the caller; this pointer is used to identify the volume during certain file system calls.

Initializing the device for use with the raw file system does not format the disk. That is done using an *ioctl( )* call with the FIODISKFORMAT function.

Since there are no file system structures on the disk, no disk initialization (FIODISKINIT) is required, although rawFs accepts such an *ioctl( )* call for compatibility with other file systems (it performs no action and always returns OK).

5.4.2.3 Mounting Volumes

A disk volume is "mounted" automatically, generally during the first *open( )* or *creat( )* operation for a file or directory on the disk. (Certain *ioctl( )* calls also cause the disk to be mounted.) When a disk is mounted, the directory data is read from the disk. Automatic mounting reoccurs on the first file access following a ready-change operation (see **5.4.2.5 Changing Disks**).

5.4.2.4 File I/O

To begin I/O to a raw file system device, the device is first opened using the standard *open*( ) function. (The *creat*( ) function can also be used, although nothing is actually "created.") Data on the raw file system device is written and read using the standard I/O routines *write*( ) and *read*( ). For more information, see **4.3 Basic I/O.**

The character pointer associated with a file descriptor (i.e., the byte offset at which reads and writes take place) can be set by using the *ioctl*( ) with the FIOSEEK function.

Multiple file descriptors can be open simultaneously for a single device. These must be carefully managed to avoid modifying data which is also being used by another file descriptor. In most cases, such multiple open descriptors should use FIOSEEK to set their character pointers to separate disk areas.

5.4.2.5 Changing Disks

The raw file system should be notified when removable disks are changed (e.g., floppies are swapped). Two different notification methods are provided and discussed below: (1) *rawFsVolUnmount*( ) and (2) the ready-change mechanism.

- **Unmounting Volumes**

The first method of announcing a disk change is to call *rawFsVolUnmount*( ) before removing the disk. This call flushes all modified file descriptor buffers if possible (see description of disk synchronization, below) and also marks any open file descriptors as obsolete. During the next I/O operation, the disk will be remounted. *rawFsVolUnmount*( ) can also be initiated by using *ioctl*( ) with FIOUNMOUNT. Any open file descriptor to the device can be used in the *ioctl*( ) call.

Subsequent attempts to use obsolete file descriptors for further I/O operations return an S_rawFsLib_FD_OBSOLETE error. To free such a descriptor, use *close*( ), as usual. This frees the descriptor, but still returns S_rawFsLib_FD_OBSOLETE.

Interrupt handlers must not call *rawFsVolUnmount*( ) directly, because it is possible for the call to pend while the device becomes available. The interrupt handler can instead give a semaphore that prepares a task to unmount the volume. (*rawFsReadyChange*( ) can be called directly from interrupt handlers, see below.)

When *rawFsVolUnmount( )* is called, it attempts to write buffered data out to the disk. Its use is therefore inappropriate for situations where the disk-change notification does not occur until a new disk is inserted – the old buffered data would be written to the new disk. In these circumstances, use *rawFsReadyChange( )* (see below).

If *rawFsVolUnmount( )* is called after the disk is physically removed, the data flushing portion of its operation will fail. However, the file descriptors are still marked as obsolete, and the disk is marked as requiring remounting. An error is *not* returned by *rawFsVolUnmount( )* in this situation. To avoid lost data in such a situation, the disk should be synchronized before it is removed (see **Synchronizing Volumes** below).

- **Announcing Disk Changes with Ready-Change**

The second method of informing **rawFsLib** that a disk change is taking place is via the *ready-change* mechanism. A change in the disk's ready status is interpreted by **rawFsLib** to indicate that the disk should be remounted during the next I/O operation.

There are three ways to announce a ready-change:

- by calling *rawFsReadyChange( )* directly
- by calling *ioctl( )* with FIODISKCHANGE
- by having the device driver set the **bd_readyChanged** field in the BLK_DEV structure to TRUE; this has the same effect as notifying **rawFsLib** directly.

The ready-change announcement does not cause buffered data to be flushed to the disk. It merely marks the volume as needing remounting. As a result, data written to files may be lost. This can be avoided by synchronizing the disk before asserting ready-change (see **Synchronizing Volumes** below). The combination of synchronizing and asserting ready-change provides all the functionality of *rawFsVolUnmount( )* except for marking file descriptors as obsolete.

Since it does not attempt to flush data or perform other operations that could cause delay, ready-change can be used in interrupt handlers.

The block device driver status-check routine (identified by the **bd_statusChk** field in the BLK_DEV structure) may be useful for asserting ready-change for devices which only detect a disk change after the new disk has been inserted. This routine is called at the beginning of each *open( )* or *creat( )* operation, before the file system checks for ready-change.

- **Disks with No Change Notification**

  If it is not possible for a ready-change to be announced each time the disk is changed, all file descriptors for the volume should be closed before the disk is changed.

- **Synchronizing Volumes**

  When a disk is *synchronized,* all buffered data that has been modified is written to the physical device so that the disk is "up-to-date." For the raw file system, the only such data is that contained in open file descriptor buffers.

  To avoid loss of data, a disk should be synchronized before it is removed. It may or may not be necessary to explicitly synchronize a disk, depending on when (or if) the *rawFsVolUnmount( )* call is issued.

  When *rawFsVolUnmount( )* is called, an attempt is made to synchronize the device before unmounting. If this disk is still present and writable at the time of the call, synchronization takes place; there is no need to synchronize the disk independently.

  However, if the *rawFsVolUnmount( )* call is made after a disk has been removed, it is too late to synchronize – *rawFsVolUnmount( )* discards the buffered data. Therefore, make a separate *ioctl( )* call with the FIOSYNC function before the disk is removed. (This could be done in response to an operator command.) Any open file descriptor to the device can be used during the *ioctl( )* call. This call causes all modified file-descriptor-buffers for the device to be written out to the disk.

5.4.2.6 I/O Control Functions Supported by rawFsLib

The raw disk file system supports the *ioctl( )* functions shown in Table 5-6 below. The functions listed are defined in the header ioLib.h. For more information, see the manual entry for rawFsLib(1) and the manual entry for *ioctl( )* in ioLib(1).

Table 5-6. I/O Control Functions Supported by rawFsLib

Function	Meaning
FIODISKFORMAT	Format the disk (device driver function).
FIODISKINIT	Initialize a raw file system on a disk volume (not required).
FIODISKCHANGE	Announce a media change.
FIOUNMOUNT	Unmount a disk volume.
FIOGETNAME	Get the file name of the *fd*.
FIOSEEK	Set the current byte offset on the device.
FIOWHERE	Return the current byte position on the device.
FIOSYNC	Write out all modified file descriptor buffers.
FIOFLUSH	Same as FIOSYNC.
FIONREAD	Get the number of unread bytes on the device.

6

Network

Contents

6

Network

6.1 Introduction

The Vx960 network is the link that connects Vx960 systems with UNIX systems and other Vx960 systems. It is compatible with the 4.3 BSD Tahoe UNIX network facilities. Vx960 also is compatible with the Network File System (NFS) developed by SUN Microsystems.

This link provides a seamless development environment between UNIX hosts and Vx960 target systems. Remote login facilities allow remote access to the Vx960 shell by users on UNIX systems, and remote access to UNIX shells by users on Vx960 systems. Remote file access allows Vx960 tasks to access UNIX files across the network. Remote procedure calls allow a task on one machine to invoke procedures which actually run on another machine.

This document gives an overview of the Vx960 network components, and the procedures necessary to configure the network.

6.2 Network Components

The hierarchy of Vx960 network components is shown in Figure 6-1. At the lowest level, Vx960 uses Ethernet as the basic transmission medium. Vx960 can also use serial lines for long-distance connections or shared memory on a common backplane in more closely coupled environments. On top of the transmission media, Vx960 uses the Internet protocols TCP/IP and UDP/IP to transport data between processes running under either Vx960 or UNIX.

Using Internet protocols, Vx960 makes several types of network facilities available to the user:

Sockets Allow communications between tasks, running either under Vx960 or UNIX.

Remote Procedure Calls
Allow a task on one machine to invoke procedures which actually run on other machines. Both the calling task and the called procedure may run under either Vx960 or UNIX.

Remote Login Allows remote access to the Vx960 from a UNIX system and remote access to a UNIX shell from a Vx960 system.

Remote File Access
Allows Vx960 tasks to access UNIX files remotely, via the Network File System (NFS), UNIX remote shell (*rsh*), or Internet File Transfer Protocol (*ftp*).

Remote Command Execution
Allows Vx960 tasks to invoke shell commands on a UNIX system, via the network.

6.2.1 Ethernet

Ethernet is one of the available mediums over which the Vx960 network operates. Ethernet is a 10 MBit/sec local area network specification that is standardized and supported by numerous vendors. It is ideal for most Vx960 applications, but there is nothing that is inherently tied to Ethernet in either the Vx960 or UNIX network systems.

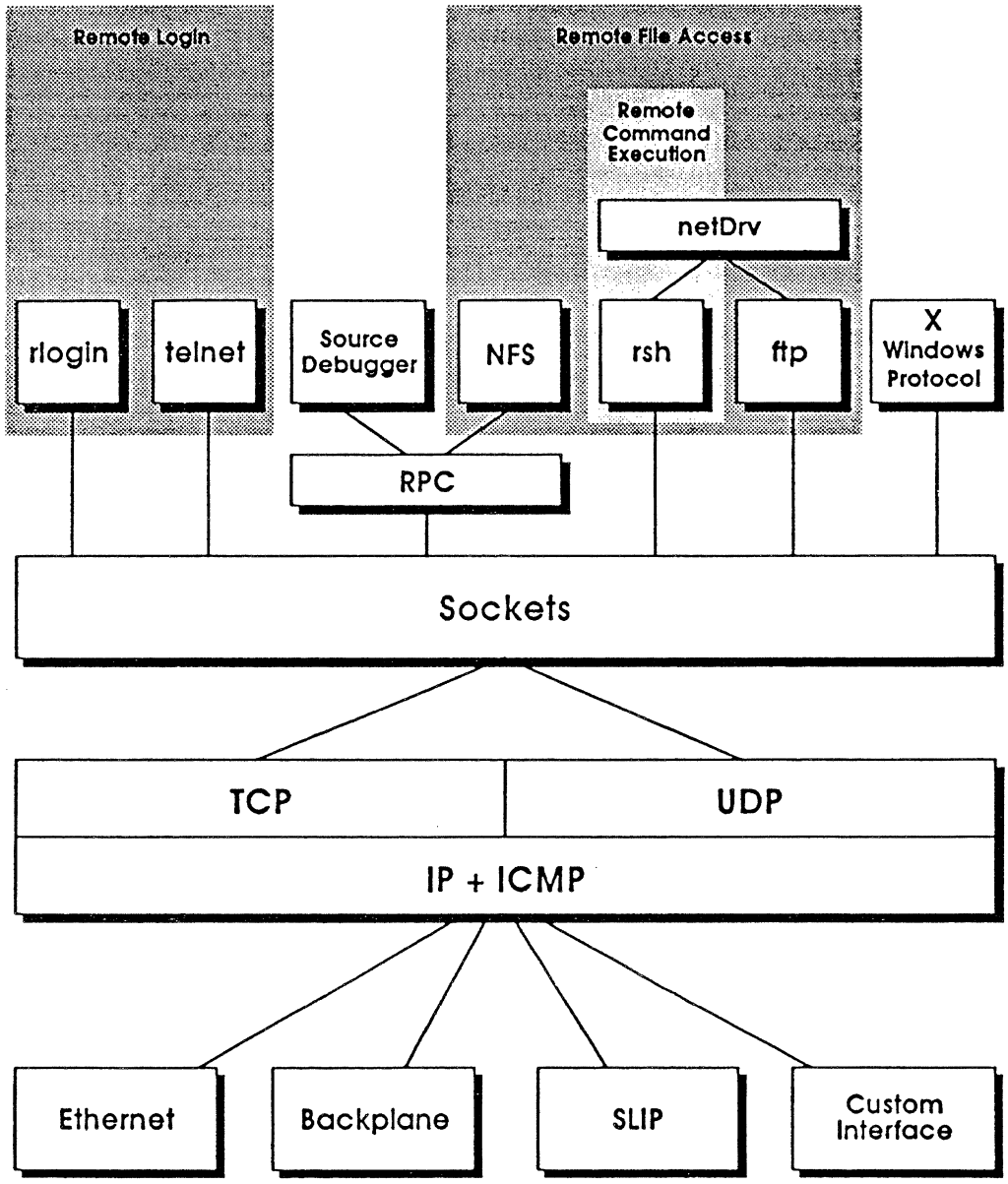


Figure 6-1. Vx960 Network Components

6.2.2 Serial Line Interface Protocol (SLIP)

The Vx960 network can communicate with the host operating system over serial connections using the Serial Line Interface Protocol (SLIP). Using SLIP as a network interface driver is a straightforward way to use TCP/IP software with point-to-point configurations such as long-distance telephone lines or RS-232 serial connections between machines.

6.2.3 Backplane Network

The Vx960 network can be used for communication among multiple processors on a common *backplane.* In this case, data is passed via shared memory. This is implemented in the form of a standard network driver so that all the higher levels of network components are fully functional over this backplane "network." Thus, all the high-level network facilities provided over Ethernet are also available over the backplane.

The backplane network allows multiple Vx960 CPUs to communicate with each other over a backplane. In addition, it allows Vx960 CPUs to be plugged directly into the backplane of some UNIX systems and to communicate with a UNIX CPU using the same shared memory scheme. In this configuration, all Vx960-to- UNIX network facilities available over Ethernet are also available over the backplane. This involves a special UNIX version of the Vx960 backplane network driver that must be configured into the UNIX system.

6.2.4 TCP/IP Internet Protocols and Addresses

6.2.4.1 Protocols

On top of the raw Ethernet and backplane transmission mechanisms, Vx960 uses the Internet protocol suite (often referred to as TCP/IP) to affect process-to-process communication across the network. Three main protocols are used:

- Internet Protocol (IP)

  IP is the base network protocol of the Internet protocol family. With IP, each *host* (computer) in the network has a unique four-byte Internet address (described below). IP accepts packets addressed to a particular host and tries to deliver

them. If multiple networks are connected by gateways, IP forwards a packet from gateway to gateway until the packet reaches a network where it can be delivered directly. IP breaks up and reassembles packets to fit the packet size of the physical network. However, IP makes no guarantees that packets will be delivered to the destination correctly. Although it is possible to access IP directly, most applications use one of the higher-level protocols such as UDP or TCP.

- User Datagram Protocol (UDP)

  UDP provides a *datagram*-based process-to-process communication mechanism. UDP extends the message address to include a *port address* in addition to the host Internet address, where a port address identifies one of several distinct destinations within a single host. Thus UDP accepts messages addressed to a particular port on a particular host, and tries to deliver them, using IP to transport the messages between the hosts. Like IP, UDP makes no guarantees that messages will be delivered correctly or delivered at all.

- Transmission Control Protocol (TCP)

  TCP provides reliable, flow-controlled, two-way, process-to-process transmission of data. TCP is a *connection*-based communication mechanism. This means that before data can be exchanged via TCP, the two communicating processes must first establish a connection via a distinct connection phase. Data is then sent and received as a byte stream at both ends. Like UDP, TCP extends the connection address to include a port address in addition to the host Internet address. That is, a connection is established between a particular port in one host and a particular port in another host. TCP *guarantees* that the delivery of data will be correct, in the proper order, and without duplication.

The Vx960 network also supports the associated Internet Control Message Protocol (ICMP) and the Ethernet Address Resolution Protocol (ARP), as implemented in UNIX BSD 4.3.

6.2.4.2 Internet Addresses

Each host in an Internet network has a unique Internet address and an associated address mask. An Internet address is 32 bits long, and begins with a Internet address class, followed by a network identifier and host identifier. The address mask is set to a default value according to class if subnets are not used. For more information, see **6.3.8 Using Subnets.**

Three classes of Internet addresses exist to accommodate different network configurations:

- Class A addresses support a small number of networks, each with a large number of hosts.
- Class B addresses support a moderate number of networks, each with a moderate number of hosts.
- Class C addresses support a large number of networks, each with a small number of hosts.

The three classes are distinguished by the high-order bits (in network byte order) of an Internet address as shown in Figure 6-2.

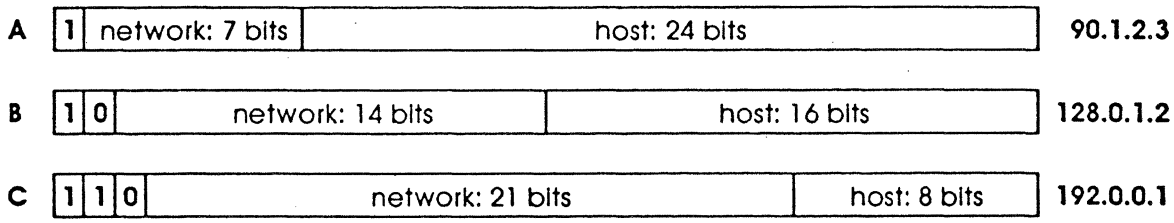


Figure 6-2 Internet Address Classes

By convention, Internet addresses are usually represented as a character string with a dot (.) notation. Dot notation lists the 32-bit number as a string of four 8-bit values separated by dots. Internally, the Internet address is often kept as a simple 32-bit value (e.g., as an *int, long, u_long, struct in_addr,* etc.). For example, the Internet address 0x5a010203 is 90.1.2.3 in standard dot notation. Each Internet address class has a unique address range determined by the high-order bits and the default address mask as shown in Table 6-1.

Vx960 includes routines for manipulating Internet addresses. For instance, there are routines for converting between dot notation and integer notation, routines for extracting network and host portions of an address, and routines for creating a new address from a network and host number. See the manual entry **inetLib**(1).

Table 6-1. Internet Address Ranges			
Class	**High Order Bits**	**Default Address Mask**	**Address Range**
A	0	0xff000000	0.0.0.0 - 126.255.255.255
Reserved			127.0.0.0 - 127.255.255.255
B	10	0xffff0000	128.0.0.0 - 191.255.255.255
C	110	0xffffff00	192.0.0.0 - 223.255.255.255

6.2.4.3 Packet Routing

The IP layer software handles packet routing. For each device connected to the network, internal routing tables contain information about possible destination addresses. These routing tables contains two types of entries: host-specific route entries and network-specific route entries. Host-specific route entries contain the host destination address and the address of the gateway to use for packets destined for this host. Network-specific route entries contain a network destination address and the Internet address of the gateway to use for packets destined for this network. An example of Internet routing is shown in Figure 6-3.

The host-specific route entries have precedence over network-specific route entries. The IP layer software first compares the destination address against any host- specific route entries in the table. If there is no matching entry, the IP layer software searches for an appropriate network-specific routing table entry. The appropriate address mask is applied to the destination address to obtain the network identifier. This network identifier is then compared against the network-specific entries in the routing table.

The Vx960 routing table is manually edited using *routeAdd*( ) and *routeDelete*( ) to add or delete route entries:

```
routeAdd ("91.0.0.0", "90.0.0.2");

routeDelete ("91.0.0.51", "90.0.0.2");
```

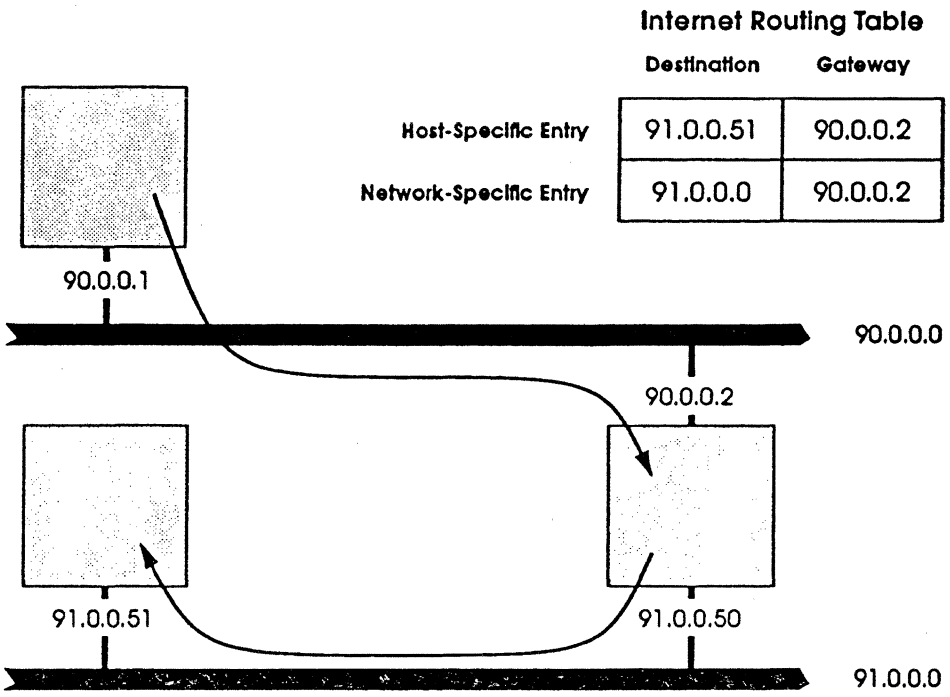


Figure 6-3. Internet Routing

6.2.5 Sockets

In Vx960, just as in many UNIX systems, the direct user interface to the Internet protocol suite is via *sockets*. A socket is an end-point for communications which gets "bound" to a UDP or TCP port within the host.

A process can create a datagram socket (UDP) and bind it to a particular port number. Other processes, on any host in the network, can then send messages to that socket by specifying the host Internet address and the port number.

Similarly, a process can create a stream socket (TCP) and bind it to a particular port number. Another process, on any host in the network, can then create another stream socket and request that it be connected to the first socket by specifying its host Internet address and port number. Once the two TCP sockets have been thus connected, there is a virtual circuit set up between them allowing error-free, socket-to-socket communications.

Socket communications is a homogeneous mechanism. Socket communications among processes is the same regardless of the location of the processes in the network, or the operating system under which they are running. Processes can communicate within a single CPU, across a backplane, across an Ethernet, or across any connected combination of networks. Socket communications can occur between Vx960 tasks and UNIX processes in any combination. In all cases, the communications look identical to the application, except for the speed of the communications.

The basic socket functions shown in Table 6-2 are provided by the library sockLib(1). Some I/O functions can be invoked on sockets.

Table 6-2. Socket Functions

Call	Description
socket( )	Create a socket.
bind( )	Bind a name to a socket.
listen( )	Listen for connections on a TCP socket.
accept( )	Accept a connection on a TCP socket.
connect( )	Initiate a connection on a socket.
shutdown( )	Shutdown a socket connection.
send( )	Send data on a socket.
sendto( )	Send a message to a socket.
sendmsg( )	Send a message on a socket.
recv( )	Receive on a socket.
recvfrom( )	Receive a datagram.
recvmsg( )	Receive a message.
setsockopt( )	Set socket options.
getsockname( )	Get socket name.
getpeername( )	Get name of connected peer socket.
read( )	Read from a socket.
write( )	Write to a socket.
ioctl( )	Perform control functions on a socket.
close( )	Close a socket.
select( )	Multiplex file I/O.

6.2.5.1 Stream Sockets (TCP)

The Transmission Control Protocol (TCP) provides reliable, guaranteed, two-way transmission of data. In a TCP communication, two sockets are connected, allowing a reliable byte-stream to flow between them in each direction. TCP is referred to as a *virtual circuit* protocol, because it behaves as though a circuit is created between the two sockets.

A good analogy for TCP communications is a telephone system. Connecting two sockets is analogous to calling from one phone to another. After the connection is established, you write and read data (talk and listen).

Table 6-3 shows the steps in establishing socket communications with TCP, and the analogy of each step with telephone communications.

Table 6-3. TCP Analogy to Telephone Communication

Task 1 Waits	Task 2 Calls	Function	Analogy
socket( )	*socket( )*	Create sockets.	Hook up telephones.
bind( )		Assign address to socket.	Assign phone numbers.
listen( )		Allow others to connect to socket.	Allow others to call.
	connect( )	Request connection to another socket.	Dial another phone's number.
accept( )		Complete connection between sockets.	Answer phone and establish connection.
write( )	*write( )*	Send data to other socket.	Talk.
read( )	*read( )*	Receive data from other socket.	Listen.
close( )	*close( )*	Close sockets.	Hang up.

6.2.5.2 Datagram Sockets (UDP)

The User Datagram Protocol (UDP) provides a simpler but less robust way of communicating. In a UDP communication, data is sent between sockets in separate, unconnected, individually addressed packets called *datagrams*.

As TCP is analogous to telephone communications, UDP is analogous to sending mail. Each UDP packet is like a letter. Each packet carries the address of both the destination and the sender. Like the mail, UDP is unreliable in that packets which are lost or out-of-sequence may not be reported.

6.2.6 Remote Procedure Calls

Remote Procedure Call (RPC) implements a client-server model of task interaction. In this model, client tasks request services of server tasks, and then wait for their reply. RPC formalizes this model and provides a standard protocol for passing requests and returning replies. Thus, a Vx960 or UNIX client task can request that services from Vx960 or UNIX servers in any combination.

Internally, RPC uses sockets as the underlying communication mechanism. RPC, in turn, is used in the implementation of several higher-level facilities, including the Network File System (NFS) and remote source-level debugging. Also, RPC includes tools to help generate the client interface routines and the server skeleton.

The Vx960 implementation of RPC was originally designed by SUN Microsystems and is in the public domain. See the public domain RPC documentation (supplied in source form in the directories **vx/pub/rpc4.0/doc** and **vx/pub/rpc4.0/man**) and the manual entry for **rpcLib**(1) for more information.

6.2.7 Remote Login

The Vx960 **rlogin** facilities allow remote access to the Vx960 shell by users on UNIX systems, and remote access to UNIX shells by users on Vx960 systems. Using the UNIX **rlogin** function, you can access the Vx960 target system's shell from a UNIX terminal or workstation. In fact, with a window-based UNIX workstation, you can access many Vx960 target systems at the same time, each in a separate window on the workstation. From Vx960, you can also log in to a UNIX shell or another Vx960 shell using *rlogin*( ).

Vx960 also allows remote access to the Vx960 shell via **telnet**. From the host computer, this works similarly to **rlogin**. However, Vx960 does not support **telnet** access from the Vx960 system to the host.

Vx960 allows only one remote login session at a time on a target system. If remote login security has been included, users must enter a user name and password when accessing the Vx960 shell across the network using **rlogin** or **telnet**. See **9.8 Remote Login**.

6.2.8 Remote File Access

Transparent remote file access allows files on remote systems to be accessed just as if they were local. Programs running under Vx960 can access files on any UNIX system, via the network, exactly as if they were local to the Vx960 system. For example, **/dk0/file** might be a file local to the Vx960 system, while **/host/file** might be a file located on another machine entirely. To programs running under Vx960, the files operate in exactly the same way; only the name is different.

Remote file access can be provided using any of three different protocols:

- **UNIX Remote Shell (rsh)**

  The UNIX *remote shell* protocol is serviced by the remote shell daemon **rshd** on UNIX. See the manual entry for **remLib**(1).

- **Internet File Transfer Protocol (ftp)**

  The Internet File Transfer Protocol (*ftp*) client and server functions are serviced by a library of routines in **ftpLib** to transfer files between *ftp* servers on the network and invoke other *ftp* functions. See the manual entry for **ftpLib**(1) and **ftpdLib**(1).

- **Network File System (NFS)**

  Developed by SUN Microsystems, the Network File System (NFS) "client" protocol is implemented in the I/O driver **nfsDrv** to access files on any NFS server on the network. This I/O driver has been tested with approximately 70 different implementations of NFS file servers on various operating systems. See the manual entry for **nfsDrv**(2).

6.2.9 Remote Command Execution

The Vx960 remote command execution facilities allow programs running on Vx960 to invoke UNIX commands and have the results returned on *standard out* and *standard error* via socket connections. This is accomplished using the UNIX *remote shell* protocol, which is serviced by the remote shell daemon **rshd** on UNIX. See the manual entry for **remLib**(1).

6.2.10 Loopback Interface

In addition to the ethernet driver, there is a software-only, network pseudo-device called the loopback interface that has the special IP address, 127.0.0.1 associated with it. If a TCP or UDP socket is bound to this address, and data is written to it, the data will be available on the local machine only; i.e., it does not actually go out on the physical network. All data sent to 127.0.0.1 is routed automatically back to the local machine by the IP network layer. This address is usually given the name "localhost". You can see "localhost" with a **hostShow** command. The loopback interface is used for software testing purposes and in some cases, local inter-process communication on UNIX or inter-task communication on the Vx960 system; e.g., two tasks on the Vx960 could use this interface to communicate via TCP without ever putting packets on the physical network. The driver associated with the loopback interface takes packets written to it and hands them back to IP as input.

6.3 Configuring the Network

Before you can use the Vx960 network, you must configure both the UNIX and the Vx960 systems. There are two main concerns in configuring the network: establishing system names and addresses, and establishing appropriate access permissions for each system.

On UNIX, most of the configuration procedures consist of setting up various network "database" files and the system startup files. In Vx960, most of the configuration information necessary for access to a single host is contained in the boot line. Further initialization can be added to **usrConfig.c**, or handled by application code, or done interactively from the shell.

The network configuration procedures for Vx960 and the host are discussed below and summarized in the **Network Procedures Summary** table on page 214.

6.3.1 Associating Inet Addresses with Network Interfaces

A system's physical connection to a network is called a *network interface.* Each network interface must be assigned a unique Internet ("inet") address. Since a system can be connected to several networks, or can even have several connections to the same network, it can have several network interfaces.

On a UNIX system, the Internet address of a network interface is specified using the **ifconfig** command:

```
% ifconfig ex0 90.0.0.1
```

This is usually done in the UNIX startup file **/etc/rc.local**. For more information, see the UNIX manual entry for **ifconfig**.

In Vx960, the Internet address of a network interface is specified by calling the routine *ifAddrSet*( ):

```
ifAddrSet ("ei0", "90.0.0.2");
```

See the Vx960 manual entry for **ifLib**(1).

The Vx960 network startup routine, *usrNetInit*( ) in **usrConfig.c**, sets the address of the interface used to boot Vx960 to the Internet address specified in the Vx960 boot parameters.

6.3.2 Associating Inet Addresses with Host Names

The underlying Internet protocol uses systems' 32-bit Internet addresses. Some users, however, prefer to use system names that are more meaningful to them. Thus both UNIX and Vx960 maintain a map between system names and Internet addresses.

On UNIX, the file /etc/hosts contains the map between system names and Internet addresses. Each line consists of an Internet address and the name(s) of the computer at that address:

```
90.0.0.2    vx1
```

There must be a line in this file for each UNIX system and for each Vx960 system in the network. For more information on /etc/hosts, see your UNIX system manual entry **hosts**(5).

In Vx960, calls to the routine *hostAdd*( ) are used to associate system names with Internet addresses. Make one call to *hostAdd*( ) for each system the Vx960 system communicates with:

```
hostAdd ("host", "90.0.0.1");
```

To associate more than one name with an Internet address, *hostAdd*( ) can be called several times with different host names and the same Internet address. The routine *hostShow*( ) displays the current system name and Internet address associations.

The Vx960 network startup routine, *usrNetInit*( ) in usrConfig.c, automatically adds the name of the host Vx960 was booted from, using the host name specified in the Vx960 boot parameters.

6.3.3 Transparent Remote File Access

As discussed previously, Vx960 can use any of three different underlying protocols to provide transparent remote file access: UNIX remote shell (*rsh*), the Internet File Transfer Protocol (*ftp*), or the Network File System (NFS).

The Vx960 I/O driver **netDrv** implements remote file access using either of the first two protocols, *rsh* or *ftp*. The **netDrv** driver uses these protocols to read the entire remote file into local memory when the file is opened and to write the file back when it is closed if it was modified.

The Vx960 I/O driver **nfsDrv** implements remote file access using NFS. This protocol transfers only the data actually read or written to the file and so is considerably more efficient, both in terms of memory utilization and throughput. However, it is somewhat more cumbersome to set up initially than the other protocols.

The following sections describe the implementation and configuration of these protocols in detail.

6.3.3.1 Transparent Remote File Access via rsh and ftp

A separate Vx960 I/O device is created for every host which services remote file accesses. When a file on one of these devices is accessed, **netDrv** uses either *rsh* or *ftp* to transfer the file to or from Vx960.

Using *rsh*, **netDrv** remotely executes the UNIX **cat** command to copy the entire requested file to and from the target. The *rsh* protocol is serviced by the remote shell daemon **rshd** on UNIX. See the manual entry **remLib(1)**.

Using *ftp*, **netDrv** uses the RETR and STOR commands to "retrieve" and "store" the entire requested file. The **netDrv** driver uses a library of routines, in **ftpLib**, that implements the "client" side for the Internet File Transfer Protocol. Vx960 tasks can transfer files to and from *ftp* servers on the network and invoke other *ftp* functions. See the manual entry **ftpLib(1)**.

Vx960 can also function as an *ftp* server. The *ftp* daemon running on a Vx960 server handles calls from UNIX and Vx960 clients, and can also boot another Vx960 system. The *ftp* server daemon is initialized on the Vx960 server using *ftpdInit( )*. See the manual entry for *ftpdInit( )* in **ftpdLib(1)**.

- **Allowing Remote File Access via rsh on UNIX**

  An *rsh* request includes the name of the requesting user. The request is treated just like a remote login by that user. UNIX restricts such remote logins via the file **.rhosts** in users' home directories, and more globally with the file **/etc/hosts.equiv**. The **.rhosts** file contains a list of system names (as defined in the **/etc/hosts** file) that have access to that user's login.

  Therefore, you should have a **.rhosts** file in your home directory listing the Vx960 systems, each on a separate line, that are allowed to access files remotely using your user name.

  The **/etc/hosts.equiv** file provides a less selective mechanism. Systems listed in this file are allowed login access to any user defined on the local system (except the super-user **root**). Thus, adding Vx960 system names to **/etc/hosts.equiv** allows those Vx960 systems to access files using any user name on the UNIX system.

  The *ftp* protocol, unlike *rsh*, specifies both the user name and password on every request. Therefore, when using *ftp*, the UNIX system does not use **.rhosts** or **/etc/hosts.equiv** to authorize remote access.

- **Creating Vx960 Network Devices That Use rsh or ftp**

  The routine *netDevCreate( )* is used to create a Vx960 I/O device for a particular remote host system:

  ```
  netDevCreate ("devName", "host", protocol);
  ```

  Its arguments are:

devName	the name of the device to be created.
host	the Internet address of the host in dot notation or the name of the remote system as specified in a previous call to *hostAdd( )*. It is traditional to make the device name be the host name followed by a colon.
protocol	the file transfer protocol: 0 for *rsh* or 1 for *ftp*.

  For example, the call:

  ```
  netDevCreate ("mars:", "mars", 0);
  ```

  creates a new I/O device on Vx960 called "mars:", which accesses files on the host system "mars" using *rsh*.

  Once a network device has been created, files on that host can be accessed by appending the host path name to the device name. For example, the file name **mars:/usr/fred/myfile** refers to the file **/usr/fred/myfile** on the "mars" system. This file can be read and/or written exactly like a local file. For example, the Vx960 shell command:

  ```
  -> fd = open ("mars:/usr/fred/myfile", UPDATE)
  ```

  would open that file for I/O access.

  The Vx960 network startup routine, *usrNetInit( )* in usrConfig.c, automatically creates a network device for the host name specified in the Vx960 boot parameters. If no *ftp* password was specified in the boot parameters, the network device is specified with the *rsh* protocol. If a password was specified, *ftp* is used.

- **Setting the User ID for Remote File Access via rsh or ftp**

  All *ftp* and *rsh* requests to a remote system include the name of the user. All *ftp* requests include a password as well as a user name. From Vx960 you can specify the user name and password for remote requests by calling the routine *iam( )*:

```
iam ("username", "password");
```

The first argument to *iam( )* is the user name which is used when you access remote systems. The second argument is the *ftp* password. This is ignored if using *rsh*, and can be specified as NULL or 0.

For example, the call:

```
iam ("fred", "flintstone");
```

tells Vx960 that all accesses to remote systems via either *rsh* or *ftp* are through user "fred", and if *ftp* is used, the password is "flintstone".

The Vx960 network startup routine, *usrNetInit( )* in usrConfig.c, initially sets the user name and password to those specified in the boot parameters.

- **File Permissions**

For a Vx960 system to have access to a particular file on a host, permissions on the host system must be set up so that the user name Vx960 is using has permission to read that file (and write it, if necessary). This means that Vx960 must have permission to access all directories in the path, as well as the file itself.

The easiest way to check this is to log in to the host with the user name that Vx960 is using, and try to read or write the file in question. If you cannot do this, neither can the Vx960 system.

6.3.3.2 Transparent Remote File Access via NFS

The I/O driver **nfsDrv** uses the "client" routines in the library **nfsLib** to access files on an NFS file server. The routines in **nfsLib** are implemented using Remote Procedure Call. For more information, see the manual entry **nfsLib**(1) and **6.2.6 Remote Procedure Calls.**

- **Allowing Remote File Access via NFS on UNIX**

To access files, NFS clients *mount* file systems from NFS servers. On a UNIX NFS server, the file **/etc/exports** specifies which of the server's file systems can be mounted by NFS clients. For example, if this file contained the line:

```
/usr
```

then the file system /usr could be mounted by NFS clients such as Vx960. If a file system is not listed in this file, then it cannot be mounted by other machines. Other optional fields in /etc/exports allow the exporting of a file system to be restricted to certain machines or users.

In addition to modifying the /etc/exports file, you must notify local servers that the file /etc/exports has been changed. You can reboot the host, or you can issue an **exportfs -a** command on the Sun, for example. See your host documentation for more details.

- **Creating Vx960 Network Devices That Use NFS**

  Access to a remote NFS file system is established by *mounting* that file system locally and creating an I/O device for it using the routine *nfsMount*( ):

  ```
  nfsMount ("host", "hostFileSys", "localName");
  ```

  Its arguments are:

host	the host name of the NFS server where the file system resides
hostFileSys	the name of the desired host file system or subdirectory
localName	the local name to assign to the file system

  For example, the following call would mount "/usr" of the host "mars" as "/vxusr" locally:

  ```
  nfsMount ("mars", "/usr", "/vxusr");
  ```

  The host name "mars" must have already been added to Vx960's list of hosts using the routine *hostAdd*( ). Vx960 then creates the local I/O device /vxusr that refers to the mounted file system. A reference on Vx960 to a file with the name /vxusr/fred/myfile refers to the file /usr/fred/myfile on the host machine "mars" as if it were local to the Vx960 system.

  Vx960 mounts all exported NFS file systems if the option INCLUDE_NFS_MOUNT_ALL is defined in the Vx960 configuration file. Otherwise, the network startup routine, *usrNetInit*( ) in usrConfig.c, tries to mount the file system Vx960 was booted from, as long as NFS is included in the Vx960 configuration and the Vx960 boot file begins with a "/". For example, if NFS is included and you boot /usr/vx/config/*target*/vxWorks, then Vx960 attempts to mount /usr from the boot host via NFS.

- **Setting the User ID for Remote File Access via NFS**

  When making an NFS request to a host system, the NFS server needs more information about the user than just the user's name. NFS is built on top of Remote Procedure Call (RPC) and uses a type of RPC authentication known as AUTH_UNIX. This authentication mechanism requires the user ID and a list of group IDs that user belongs to.

  These parameters can be set on Vx960 using the routine *nfsAuthUnixSet( )*:

  ```
  nfsAuthUnixSet ("mars", 1000, 200, 0);
  ```

  This would set the NFS authentication to user ID 1000 with the single group ID 200. The routine *nfsAuthUnixPrompt( )* provides a more interactive way of setting the NFS authentication parameters from the shell.

  On UNIX systems, a user ID is specified in the /etc/passwd file. A list of groups that a user belongs to is specified in the /etc/group file.

  A default user ID and group ID is specified in the Vx960 configuration header /usr/vx/config/all/configAll.h by defining the values of NFS_USER_ID and NFS_GROUP_ID respectively. The NFS authentication parameters is set to these values at system startup.

6.3.4 Remote Login from UNIX to Vx960: rlogin and telnet

You can log in to a Vx960 shell from a UNIX terminal or workstation using either **rlogin** or **telnet**. These facilities are initialized on Vx960 by the routines *rlogInit( )* and *telnetInit( )*, respectively.

The Vx960 network startup routine, *usrNetInit( )* in usrConfig.c, automatically initializes either **rlogin** or **telnet**. If no password is specified in the boot parameters, then **rlogin** automatically is initialized. If a password is specified in the boot parameters, then **telnet** automatically is initialized.

If remote login security has been included, users are required to enter a user name and password when accessing the Vx960 shell across the network using **rlogin** or **telnet**. For more information, see **9.8 Remote Login**.

6.3.5 Remote Login from Vx960 to UNIX: *rlogin( )*

You can log in to a UNIX shell from a Vx960 terminal using ***rlogin( )***. To do this, access permission must be granted to the Vx960 system by entering its system name either in the **.rhosts** file (in the user home directory) or in the **/etc/hosts.equiv** file. For more information, see **Allowing Remote File Access via rsh on UNIX** under section 6.3.3.1.

6.3.6 Adding Gateways to a Network

The Internet protocols allow hosts on different but connected networks to communicate. If a machine on one network sends a packet to a machine on another network, then a *gateway* is sought that can forward the message from the sender's network to the destination's network. If a system has interfaces to more than one network, then it can be a gateway between those networks. One of the primary functions of IP (the lower-level protocol of TCP/IP) is to perform this routing and forwarding among interconnected networks.

Many systems, including most UNIX systems, have a routing daemon (**routed**) which exchanges routing information with other systems to determine network connectivity. Vx960, however, has no routing daemon, and must instead be told explicitly about any gateways it may need. Similarly, if Vx960 systems are gateways, the UNIX systems must be told about them, since the Vx960 systems do not broadcast their routing information.

6.3.6.1 Adding a Route on UNIX

A UNIX system can be told explicitly about a gateway in one of two ways: by editing /etc/gateways or using the **route** command.

When the UNIX route daemon **routed** is started (usually at boot time), it reads a static routing configuration from /etc/gateways. Each line in /etc/gateways specifies a network gateway in the following format:

net *destination-addr* **gateway** *gateway-addr* **metric** *n* **passive**

where *n* is the "hop count" from the host machine to the destination network (i.e., the number of gateways between the host and the destination network).

For example, consider a system on network 90. The following line in /etc/gateways describes a gateway between networks 90 and 91, with an Internet address 90.0.0.3 on network 90; a hop count (metric) of 1 specifies that the gateway is a direct connection between the two networks:

```
net 91.0.0.0 gateway 90.0.0.3 metric 1 passive
```

After editing /etc/gateways, you must kill the route daemon and restart it, since it only reads /etc/gateways when it starts, and otherwise is unaware of subsequent changes to the file.

Alternatively, you can use the route command to add routing information:

route add *destination-network gateway-addr* [*metric*]

For example, the command:

```
route add 91.0.0.0 90.0.0.31
```

configures the gateway as in the above example with the /etc/gateways file. Routes added with this "manual" method are lost the next time the system is rebooted.

6.3.6.2 Adding a Route on Vx960

To add gateways to the Vx960 network routing tables, use the *routeAdd*( ) routine:

routeAdd ("*destination-addr*", "*gateway-addr*")

The arguments to the *routeAdd*( ) routine are:

destination-addr the address you want to reach

gateway-addr the address of a gateway

In other words, use *gateway-addr* to reach *destination-addr*. Both addresses can be specified by dot notation or by the host names defined by the routine *hostAdd*( ).

For example, consider two Vx960 machines *vx2* and *vx3*, both interfaced to network 91. Suppose that *vx3* is a gateway between networks 90 and 91 and that its Internet address on network 91 is 91.0.0.3. The following calls can then be made on *vx2* to establish *vx3* as a gateway to network 90:

```
routeAdd ("90.0.0.0", "vx3");
```

or

```
routeAdd ("90.0.0.0", "91.0.0.3");
```

Other routing routines are available in the routing library routeLib(1).

The Vx960 network startup routine, *usrNetInit( )* in usrConfig.c, automatically adds the gateway specified in the boot parameters (if any) to the routing tables. In this case, the address specified in the gateway field ("g = ") is added as the gateway to the network of the boot host.

6.3.7 Broadcast Addresses

Many physical networks support the notion of *broadcasting* a packet to all hosts on the network. A special Internet *broadcast address* is interpreted by the network subsystem to mean "all systems" when specified as the destination address of a datagram message (UDP). An example of this can be found in the demo program /usr/vx/demo/dg/dgTest.c.

Unfortunately there is some ambiguity about what address is to be interpreted as the broadcast address. The Internet specification now states that the broadcast address is an Internet address with a host part of all ones. However, some older systems use an Internet address with a host part of all zeros as the broadcast address.

Most newer systems, including Vx960, *accept* either address on incoming packets as being a broadcast packet. But when an application wants to *send* a broadcast packet, it must use the correct broadcast address for its system.

Vx960 uses a host part of all ones as the broadcast address. Thus a datagram sent to Internet address 90.255.255.255 (0x5affffff) would be broadcast to all systems on network 90. However, to allow compatibility with other systems, Vx960 allows the broadcast address to be reassigned for each network interface by calling the routine *ifBroadcastSet( )*. Similarly, most UNIX systems have a broadcast option to the ifconfig command that allows the broadcast address for an interface to be reassigned. See the Vx960 manual entry for *ifBroadcastSet( )* and the UNIX manual entry for ifconfig for more information.

6.3.8 Using Subnets

An Internet address consists of a network address portion and a host address portion. As described previously, there are different classes of Internet addresses in which different parts of the 32-bit address are assigned to each portion. This provides a great deal of flexibility in network addressing. Even so, in some environments network addresses are a scarce resource – an organization may be limited to certain number of network addresses by a higher authority.

To allow a single network address to be subdivided into multiple sub-networks, a technique called *subnet addressing* is used. This technique involves extending the network portion of the addresses used on a particular set of physical networks. The interpretation of the Internet address is altered to include more bits in the network portion and fewer in host portion.

The specification of which bits are to be interpreted as network address is called the *net mask*. A net mask is a 32-bit value with 1's in all bit positions that are to be interpreted as the network portion. In Vx960, the net mask can be specified for a particular network interface with the routine *ifMaskSet*( ). See the manual entry for *ifMaskSet*( ) for more information.

It may be necessary to specify a net mask during booting, in order to correctly access the host from which you are booting. This can be done by appending *":mask"* to the Internet address specifications for the Ethernet and/or backplane interfaces in the boot parameters, where *mask* is the desired net mask in hexadecimal. For example, when entering boot parameters interactively:

```
inet on ethernet (e):   90.0.0.1:ffffff00
inet on backplane (b):  90.0.1.1:ffffff00
```

or when specifying the boot parameters in a boot string:

```
e=90.0.0.1:ffffff00 b=90.0.1.1:ffffff00
```

6.3.9 Network Management and Information

The following Vx960 statistical reporting features allow you to analyze the status of the network.

Note: If you do not understand the information provided by the show commands, refer to your local network expert.

ifShow( )	displays the attached network interfaces. This is similar to the BSD **ifconfig** command.
icmpstatShow( )	displays ICMP protocol statistics. This is similar to the BSD **netstat -s** command.
inetstatShow( )	displays active connections for Internet protocol sockets. This is similar to the BSD **netstat -a** command.
ipstatShow( )	displays IP protocol statistics. This is similar to the BSD **netstat -s** command.
mbufShow( )	reports mbuf (network buffer) statistics. This is similar to the BSD **netstat -m** command.
tcpstatShow( )	displays all statistics for the TCP protocol. This is similar to the BSD **netstat -s** command.
udpstatShow( )	displays statistics for the UDP protocol. This is similar to the BSD **netstat -s** command.
arptabShow( )	displays the current ARP table entries. This is similar to the BSD **arp -a** command.

Refer to **Chapter 8, Configuration,** to make sure that these statistical reporting features are configured into your kernel.

6.3.9.1 Ping Messages From a Host

The Vx960 system responds to **ping** messages from a host. These messages tell you that a system is up and responding to at least the IP/ICMP protocol levels. Another common thing to do with **ping** is to send a number of packets and see if any of the packets are lost. This loss might indicate cable or other hardware problems, e.g., on a Sun host, you might do the following:

```
sun> /usr/etc/ping phobos
```

See the **ping** man page on your host for additional information.

Network Procedures Summary

Function	On UNIX	On Vx960
Associate Internet addresses with network interfaces.	Use ifconfig in/etc/rc.loc: `ifconfig ex0 90.0.01` or: `ifconfig ex0 host`	Call *ifAddrSet*: `ifAddrSet ("ei0", "90.0.0.2");`
Associate Internet addresses with system names.	Add address-name pairs to /etc/hosts and one of these: `90.0.0.1 host` `90.0.0.2 vx1 sonny` `90.0.0.3 vx3`	Call *hostAdd*( ): `hostAdd ("host", "90.0.0.1");` `hostAdd ("vx1", "90.0.0.2");` `hostAdd ("sonny", "90.0.0.2");`
Examine host names.	Look at `/etc/hosts`.	Call *hostShows*( ).
Transparent remote file access via *rsh*.	Add remote system names to /etc/hosts.equiv or */userhome/*.rhosts: `vx1` `vx2`	Create network devices to remote systems using *rsh*: `netDevCreate("host:","host", 0);` Set user name using *iam*( ): `iam ("fred", 0);` Access files with created device name: `fd = open ("host:/usr/myfile", UPDATE);`
Transparent remote file access via *ftp*.	No action necessary.	Create network devices to remote systems using *ftp*: `netDevCreate ("host:", "host", 1);` Set user name and password using *iam*( ): `iam ("fred", "flintstone");` Access files with created device name: `copy < host:/usr/fred/myfile`
Transparent remote file access via NFS.	Add the names of mountable file systems and a list of groups that have access to them in /etc/exports.	Create NFS device `nfsMount ("host", "/usr", "/hostUsr");` Set NFS authentication with *nfsAuthUnixSet*( ) or *nfsAuthUnixPrompt*( ): `nfsAuthUnixSet ("host", uid, gid, 0);`

Network Procedures Summary (continued)

Function	On UNIX	On Vx960
		Access files with mounted name: `copy < /hostUsr/fred/myfile`
Remote login from UNIX to Vx960 via rlogin.	Use rlogin: `% rlogin vx`	Initialize rlogin: `rlogInit ();` (Automatically called in usrConfig.c.)
Remote login from UNIX to Vx960 via telnet.	Use telnet: `% telnet vx`	Initialize telnet: `telnetInit ();` (Automatically called in usrConfig.c.)
Remote login UNIX via *rlogin*( ).	Add remote system names to /etc/hosts.equiv */userhome/*.rhosts: `vx1`	Set user name with *iam*( ): `iam ("fred");` Use *rlogin*( ): `-> rlogin "host"`
Add gateways to a network.	Add gateway to /etc/gateways and restart routed: `% net 91.0.0.0 gateway 90.0.0.3 \` `metric 1 passive` or use route: `% route add 91.0.0.0 90.0.0.3 1`	Call *routeAdd*( ): `routeAdd ("90.0.0.0", "vx3");` `routeAdd ("90.0.0.0","91.0.0.3")`
Examine routing tables.	Use netstat: `% netstat -r`	Call *routeShow*( ).
Examine network interfaces.	Use ifconfig: `% ifconfig le0`	Call *ifShow*( ): ifShow ("ei0");
Examine arp tables.	Use arp -a	Call *arptabShow*.
Add arp entry (with Intel arpLib.o).	Use `arp -s 90.0.0.3` `0:80:f9:1:2:3 temp`	Call `arpAdd ("90.0.0.3"` `"0:80:f9:1:2:3", 0)`

6.4 Network Initialization on Startup

Most of the information that Vx960 needs to set up its network and access its boot host is taken from the boot parameters you supply to the Vx960 boot ROMs via the boot line or the boot menu commands. This section summarizes the network configuration performed automatically by Vx960, based on these parameters. Most of this configuration is done by the routine *usrNetInit( )* in the configuration module /usr/vx/config/all/usrConfig.c. See **8. Configuration** for more information on usrConfig.

Vx960 startup procedures configure the network based on the following boot parameters:

boot device — The network device to boot from, e.g., "ei" for Intel 82596 coprocessor. This device is attached and configured automatically with the correct Internet address.

host name — The name of the host to boot from. This need not be the same name used internally by that system. Vx960 adds the host name to the host table and creates a device by that name.

host inet — The Internet address of the host to boot from.

inet on ethernet — The Internet address of this target on the Ethernet, if any. If the target has no Ethernet controller (perhaps because it boots from a backplane network through a gateway) this field should be blank. A subnet mask can also be specified as described previously.

inet on backplane — The Internet address of this target on the backplane network. This field can be blank if no backplane network is required. Again, a subnet mask can be specified as described previously.

gateway inet — The Internet address of the gateway to boot through, if the host is not on the same network as the target.

boot file — The full path name of the Vx960 object module to be booted.

processor number — The backplane processor number of the target CPU. The first CPU must be processor number 0.

The following network elements are configured from the above parameters:

ethernet interface If "Inet On Ethernet" is specified, the Ethernet interface is attached; the Internet address and the optional net mask are set (*ifAddrSet*( ) and *ifMaskSet*( )).

backplane interface
If "Inet on Backplane" is specified, the backplane interface is attached; the Internet address and the optional net mask are set (*ifAddrSet*( ) and *ifMaskSet*( )).

host names Host name entries are added (*hostAdd*( )) for the specified boot host and for loopback ("localhost").

routing If a gateway address is specified, a routing entry is added indicating that address is a gateway to the network of the specified boot host.

remote file access device
This device is created with the name "*boothost:*". If a password is specified, *ftp* is used. Otherwise, *rsh* is used.

network file system
If NFS is included, a mount of the first directory in the boot file path is attempted if its name begins with a "/". For example, if you boot /usr/vx/config/*target*/vxWorks, Vx960 attempts an NFS mount of /usr. If INCLUDE_NFS_MOUNT_ALL is defined, Vx960 mounts all exported NFS file systems. The NFS user ID and group ID are initialized to values defined in /usr/vx/config/all/configAll.h.

remote login rlogin is initialized if no password was specified, otherwise telnet is initialized.

user name and password
These are initialized as specified in the boot parameters.

current working directory
This is set to the remote file access device "*boothost:*".

6.5 Serial Line Interface Protocol

Vx960 can communicate with the host operating system over serial connections as well as over networks and backplanes. The Serial Line Interface Protocol (SLIP) supports TCP/IP layer software where high-speed networks are not feasible like long-distance telephone lines. If either end of a SLIP connection has other network interfaces (e.g., Ethernet) and can forward packets to other machines, a SLIP connection can serve as a gateway between networks.

Implementing SLIP as a network interface driver provides a ready connection to the IP layer software. The SLIP driver treats IP packets as streams of data suitable for serial transmission.

Since a SLIP connection is point-to-point, a pair of valid Internet addresses must be specified to inform the IP layer about how to route IP packets between the two end points of a serial line connection. The SLIP device initialization routine *slipInit*( ) contains the Internet addresses for both ends of the SLIP point-to-point connection and the name of a serial I/O device used for full-duplex communications.

For example,

```
slipInit (0, "/tyCo/1", "192.10.1.1", "192.10.1.2", 0);
```

initializes a SLIP device which uses the console's second port whose Internet address is 192.10.1.1 and whose remote peer's address is 192.10.1.2.

6.6 Backplane Networks

The Vx960 network subsystem has many layers of protocols. At the bottom layer are the network interface drivers whose job is to transmit and receive packets on the physical network medium. In addition to supplying drivers for traditional network media such as Ethernet, Vx960 also supplies a *backplane driver,* bp, which provides communication over a backplane bus.

The advantage of a backplane driver compatible with the rest of the network subsystem is that all higher-level protocols are immediately available over the back-

plane, just as they are over Ethernet. Socket communications, remote login, remote file access, NFS, remote procedure calls, and remote source debugging are all available to and from any processor on the backplane, simultaneously. Use of the network facilities over the backplane is indistinguishable from use over any other medium.

A multiprocessor backplane bus becomes an Internet network of its own. Each backplane network has its own network number. As usual, each processor on the backplane has a unique Internet address.

The routing capabilities of the Vx960 Internet protocols allow the processors on the backplane network to reach systems on other networks via a *gateway* processor on the backplane. The gateway processor has connections to both the backplane network and an external network, typically an Ethernet network. This makes all levels of network communications available between any processor on the backplane and any other host or target system on the external network.

Finally, the low-level transport mechanism of the backplane driver is also available directly. This allows alternative protocols to be run over the backplane in parallel with the standard ones.

The Vx960 backplane driver uses the following techniques to send network packets from one processor on the backplane to another:

- Packets are transferred across the backplane via a pool of *shared memory* that can be accessed by all processors on the backplane.
- Access to the shared memory pool is interlocked by use of a "test-and-set" instruction.
- Processors can either poll the shared memory data structures for input packets periodically or be notified of input packets by interrupts.

The backplane network is configured by constants in the configuration header **config.h** and by parameters specified to the Vx960 boot ROMs. The following sections give the details of the backplane network operation and configuration.

6.6.1 The Backplane Shared Memory Pool

The basis of the Vx960 backplane network is the *shared memory pool.* This is a contiguous block of memory that must be accessible to all processors on the backplane. This memory is either part of one of the processors' on-board, dual-ported memory, or on a separate memory board.

6.6.1.1 Backplane Processor Numbers

The processors on the backplane are each assigned a unique *backplane processor number* starting with 0. The assignment of numbers is arbitrary, except for processor 0, which by convention is the backplane master, described in the next section.

The processor numbers are established when the system is booted by the parameters supplied to the boot ROMs. These parameters can be burned into ROM, set in the processor's nonvolatile RAM if available, or entered interactively.

6.6.1.2 The Backplane Master: Processor 0

One of the processors on the backplane is the *backplane master.* The backplane master has the following responsibilities:

- Initializing the shared memory pool and the *backplane anchor.*
- Maintaining the *backplane heartbeat.*
- Functioning (usually) as the gateway to the external (Ethernet) network.
- Allocating the shared memory pool itself from its dual-ported memory, in some configurations.

No processor can use the backplane network until the master has initialized it. However, the master processor is *not* involved in the actual transmission of packets on the backplane between other processors. Once the shared memory pool is initialized, the processors, including the master, are all peers.

The configuration module **usrConfig.c** is set up to establish backplane processor 0 as the master. The master processor usually boots from the external (Ethernet) network directly. The master has two Internet addresses in the system: its Internet address on the Ethernet and its address on the backplane network. See the manual entry for usrConfig(1).

The other processors on the backplane boot indirectly via the backplane network, using the master as the gateway. They have only an Internet address on the backplane network. These processors specify the backplane interface, *bp*, as the boot device in the boot parameters.

6.6.1.3 The Backplane Anchor

In various configurations, the shared memory pool may be located at different locations. In many situations, it is desirable to allocate the shared memory at run-time, rather than fixing its location at the time the system is built.

All processors on the backplane need to be able to locate the shared memory pool, even when its location is not known at the time the system is built. The *backplane anchor* serves as a common point of reference for all processors. The anchor is a small data structure located at a fixed location at the time the system is built. This is usually either in low memory of the dual-ported memory of one of the processors, or at some fixed address on the separate memory board.

The anchor contains a pointer to the actual shared memory pool. This is set up by the backplane master when the backplane network is initialized. The anchor's "pointer" to the shared memory pool is actually a relative offset from the anchor itself, so the anchor and pool must be in the same address space, so that this offset is the same for all processors.

The backplane anchor address is established in two ways: either by parameters in config.h, or by boot parameters. For the backplane master, the anchor address is established in the master's configuration header config.h at the time the system image is built. The value of the constant BP_ANCHOR_ADRS, in config.h of the backplane master, should be set to the address of the anchor *as seen by the master.*

For the other processors on the backplane, a default anchor address can be established in the same way, by the setting of BP_ANCHOR_ADRS in config.h, but this requires burning boot ROMs with that configuration since the other processors must boot from the backplane to begin with. For this reason, the anchor address can also be specified in the boot parameters, if the backplane network is the boot device. This is done by appending the address to the backplane boot device code "bp", separated by "=". Thus the following boot parameter:

```
boot device:  bp=800000
```

establishes the anchor address at 0x800000. In this case, this should be the address of the anchor *as seen by the processor being booted.*

6.6.1.4 The Backplane Heartbeat

The processors on the backplane must not attach to the backplane network until the shared memory pool initialization has been completed. To let the other processors know when the backplane is "alive," the master maintains a backplane heartbeat. This heartbeat is a value in the second long word of the anchor data structure that is incremented by the master once per second. Processors on the backplane determine that the backplane is alive by watching the heartbeat for a few seconds.

6.6.1.5 Shared Memory Location

As mentioned above, the backplane shared memory can either be located at a fixed location at the time the system is built, or be dynamically allocated at run-time. The location is determined by the value of the constant BP_MEM_ADRS in the header config.h. This constant can be specified as follows:

- Specifying "NONE" (-1) means that the shared memory pool is to be dynamically allocated from the master's on-board dual-ported memory.
- Specifying an absolute address that is *different* from the anchor address BP_ANCHOR_ADRS means that the shared memory pool starts at that fixed address.
- For convenience, specifying the *same* absolute address as the anchor address means the shared memory pool starts immediately after the anchor data structure, without having to know the size of that structure.

6.6.1.6 Shared Memory Size

The size of the shared memory pool is determined by the value of the constant BP_MEM_SIZE in the header file config.h.

The size required for the shared memory pool is dependent on the number of processors and the expected traffic. There is less than 2 Kbytes of overhead for data structures. After that, the shared memory pool is divided into 2-Kbyte packets. Thus, the maximum number of packets that can be outstanding on the backplane

network is (*poolsize* - 2K) / 2K. A reasonable minimum is 64K. A configuration with a large number of processors on one backplane and many simultaneous connections could require 512 Kbytes. Having too small a pool slows down communications.

6.6.1.7 "On-Board" and "Off-Board" Options

The **config.h** files delivered with Vx960 contain a conditional compilation that makes it easy to select a pair of typical configurations. The constant BP_OFF_BOARD can be defined TRUE to select a typical *off-board* shared memory pool, or FALSE to select a typical *on-board* shared memory pool.

The *off-board* configuration establishes the backplane anchor and pool to be located at an absolute address of 0x800000 on a separate memory board with a size of 512 Kbytes.

The *on-board* configuration establishes the backplane anchor at a low address in the master processor's dual-ported memory. The shared memory pool is configured to be allocated from the master's own memory (with *malloc*( )) at run time. The size of the pool allocated is set to 64K.

These configurations are provided as examples and should be changed to suit the actual configuration.

6.6.1.8 Test-and-Set to Shared Memory

Unless some form of mutual exclusion is provided, multiple processors simultaneously accessing certain critical data structures of the shared memory pool could interfere with each other's access and cause fatal errors. The Vx960 backplane network uses an indivisible, atomic-modify instruction to obtain exclusive use of a shared memory data structure. This translates into a *read-modify-write* (RMW) cycle on the backplane bus.

It is important that the selected shared memory support the RMW cycle on the bus and guarantee the indivisibility of such cycles. This is especially problematic if the memory is dual-ported, as the memory must then also lock out one port during a RMW cycle on the other.

Some processors do not support RMW indivisibly in hardware, but do have software hooks to allow this. For example, some processor boards have a flag that can be set to prevent the board from releasing the backplane bus, once it has been

acquired, until that flag is cleared. These techniques can be implemented in the system-dependent library **sysLib.c** for the processor, in the routine *sysBusTas( )*. The backplane driver calls this routine to affect the mutual exclusion on shared memory data structures.

6.6.2 Inter-Processor Interrupts

Each processor on the backplane has a single *input queue* of packets sent from other processors to that destination processor. There are three methods processors use to determine when to examine the input queue: polling, bus interrupts, and mailbox interrupts.

When using polling, the processor simply examines its input queue periodically. When using interrupts, the processor receives an interrupt from the sending processor when its input queue has packets. Interrupt-driven communication is much more efficient than polling.

Most backplane busses have a limited number of bus-interrupt lines available on the backplane (e.g., VMEbus has seven). A processor can use one of these interrupt lines as its input interrupt. However, each processor must have its own interrupt line. Furthermore, not all processor boards are capable of generating bus interrupts. Thus, bus interrupts are difficult to use.

mailbox interrupts, also called *location monitors*, monitors the access to specific memory locations. A mailbox interrupt is a bus address which, when written to, causes a specific interrupt on the processor board. Each board can be set, with hardware jumpers or software registers, to use a different address for its mailbox interrupt.

To generate a mailbox interrupt, a processor writes to that location. There is effectively no limit to the number of processors that can use mailbox interrupts because each processor takes up a single address on the bus. Most modern processor boards include some kind of mailbox interrupt.

Each processor must tell the other processors what method to use to notify it when its input queue has packets. In the shared memory data structures, each processor enters its *interrupt type* and up to three parameters about that type. This information is used by the backplane driver of the other processors when sending packets to that processor.

The interrupt type and parameters for each processor are specified in **config.h** by the constants BP_INT_TYPE and BP_INT_ARG*n*. The possible values of BP_INT_TYPE and

the corresponding parameters are defined in the header **if_bp.h**. They are summarized in Table 6-4.

Table 6-4. Backplane Interrupt Types

Type	Arg 1	Arg 2	Arg 3	Meaning
BP_INT_NONE	-	-	-	Polling
BP_INT_BUS	level	vector	-	Bus interrupt
BP_INT_MAILBOX_1	address space	address	value	1-byte write mailbox
BP_INT_MAILBOX_2	address space	address	value	2-byte write mailbox
BP_INT_MAILBOX_4	address space	address	value	4-byte write mailbox
BP_INT_MAILBOX_R1	address space	address	-	1-byte read mailbox
BP_INT_MAILBOX_R2	address space	address	-	2-byte read mailbox
BP_INT_MAILBOX_R4	address space	address	-	4-byte read mailbox

6.6.3 Configuring the UNIX Host

The UNIX host is configured to support the backplane network using the procedures outlined earlier in this chapter for non-backplane networks. In particular, a backplane network requires:

- all backplane host names and addresses entered in /etc/hosts
- all backplane host names entered in **.rhosts** in your home directory or in **/etc/hosts.equiv**
- a gateway entry specifying the master's Internet address on the Ethernet as the gateway to the backplane network.

6.6.4 Example Configuration

This section pulls the foregoing discussion together in an example of a simple backplane network. The configuration consists of a single UNIX host and two target processors on a single backplane. In addition to the two processors, the backplane also has a separate memory board for the shared memory pool and an Ethernet controller board. The additional memory board is not essential, but makes for a configuration that is easier to describe.

In this configuration, shown in Figure 3, we have two networks: the Ethernet and the backplane. The Ethernet is assigned network number 90 and the backplane is assigned 91.

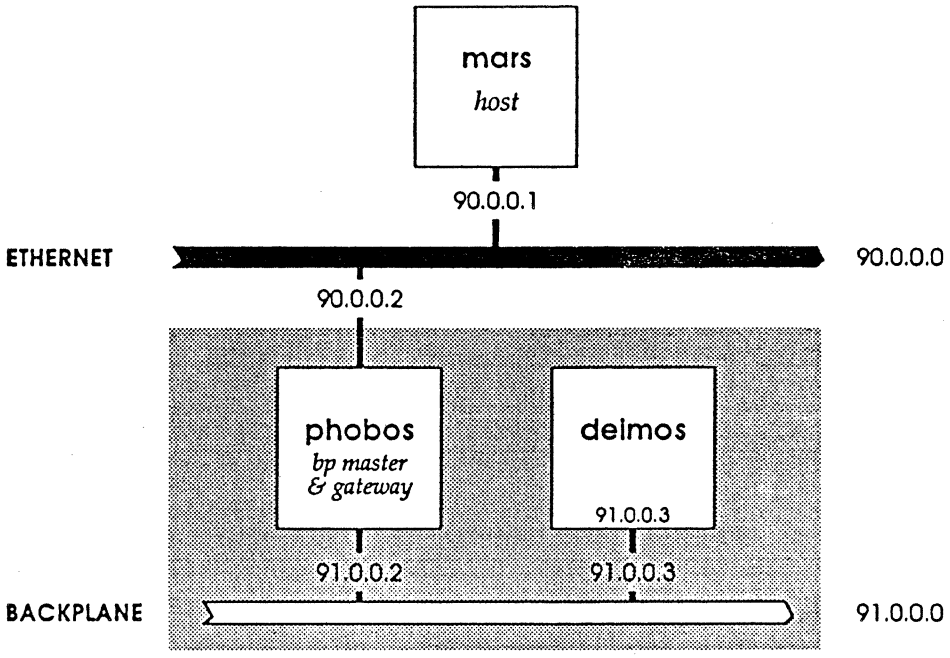


Figure 6-4. Example Backplane Network

The host is "mars" and is assigned the Internet address 90.0.0.1.

The backplane master is "phobos" and functions as the gateway between the Ethernet and backplane networks. It has two Internet addresses: 90.0.0.2 on the Ethernet network and 91.0.0.2 on the backplane network.

The other backplane processor is "deimos" and is assigned the backplane address 91.0.0.3. It has no address on the Ethernet because it is not, in fact, on the Ethernet. However, it can communicate with "mars" via the backplane network, using "phobos" as a gateway. Of course, the use of the gateway is handled by the Internet protocol and is completely transparent to the user.

The example network address assignments are as follows:

Name	Inet on Ethernet	Inet on Backplane
mars	90.0.0.1	-
phobos	90.0.0.2	91.0.0.2
deimos	-	91.0.0.3

To configure the UNIX system in our example, the /etc/hosts file must contain the Internet address and name of each system. The backplane master has two entries. The second entry, "phobos.bp", is not really necessary since the UNIX system never accesses that system with that address, but it is useful to include it in the file.

The entries in /etc/hosts are as follows:

```
90.0.0.1    mars
90.0.0.2    phobos
91.0.0.2    phobos.bp
91.0.0.3    deimos
```

To allow remote access of the UNIX host from the target systems, the .rhosts file in the users home directory, or the file /etc/hosts.equiv, must contain their names:

```
phobos
deimos
```

To inform the UNIX system of the existence of the Ethernet-to-backplane network gateway, the following line should be in the file /etc/gateways, *at the time the route daemon* routed *is started.*

```
net 91.0.0.0 gateway 90.0.0.2 metric 1 passive
```

Alternatively, you can add the route manually with the following UNIX command:

```
% route add 91.0.0.0 90.0.0.2 1
```

The target systems are configured in part by the parameters shown below in the configuration header config.h:

Parameter	Value	Comment
BP_ANCHOR_ADRS	0x800000	Address of anchor as seen by phobos.
BP_MEM_ADRS	0x800000	Address of shared memory pool as seen by phobos.
BP_MEM_SIZE	0x80000	Size in bytes of shared memory pool.
BP_INT_TYPE	BP_INT_MAILBOX_1	Interrupt targets with 1-byte mailbox.
BP_INT_ARG1	VME_AM_SUP_SHORT_IO	Mailbox in short I/O space.
BP_INT_ARG2	(0xc000 | (sysProcNum * 2))	Mailbox at: 0xc000 for phobos 0xc002 for deimos
BP_INT_ARG3	0	Write 0 value to mailbox.

The backplane master "phobos" have the following boot parameters:

```
boot device                      : enp
host name                        : mars
file name                        : /usr/vx/config/target/vxWorks
inet on ethernet (e)             : 90.0.0.2
inet on backplane (b)            : 91.0.0.2
host inet (h)                    : 90.0.0.1
gateway inet (g)                 :
user (u)                         : fred
ftp password (pw) (blank=use rsh) :
processor number                 : 0
flags (f)                        : 0
```

The other target "deimos" have the following boot parameters:

```
boot device                          : bp=800000
host name                            : mars
file name                            : /usr/vx/config/target/vxWorks
inet on ethernet (e)                 :
inet on backplane (b)                : 91.0.0.3
host inet (h)                        : 90.0.0.1
gateway inet (g)                     : 91.0.0.2
user (u)                             : fred
ftp password (pw) (blank=use rsh)    :
processor number                     : 1
flags (f)                            : 0
```

6.6.5 Troubleshooting

If you have trouble configuring a backplane network, here are a few troubleshooting procedures you can use.

1. Boot a single processor in the backplane without any additional memory or processor cards. Omit the *inet on backplane* parameter to prevent the processor from trying to initialize the backplane.

2. Power off and add the memory board, if you are using one. Power on and boot the system again. With the Vx960 boot ROM commands for display memory (**d**) and modify memory (**m**), verify that you can access the shared memory at the address you expect, with the size you expect.

3. Reboot the system, this time filling in the *inet on backplane* parameter. This parameter causes Vx960 to initialize the backplane. While booting, you see the message:

   ```
   Attaching backplane interface bp0 at anchor-addrs. done.
   ```

4. When Vx960 is up, you can display the state of the backplane network by typing:

   ```
   -> bpShow ["interface"] [, 1]
   ```

   The interface parameter is "bp0" by default. *bpShow*( ) displays cumulative activity statistics; specifying 1 as the second argument resets totals to zero.

5. Now power off and add the second processor board. Remember that the second processor must *not* be configured to be the system controller board. Power

on and stop the second processor from booting by typing any key to the boot ROM program. Boot the first processor as you did before.

6. If you have trouble booting the first processor with the second processor plugged in, you have a hardware conflict. Check that only the first processor board is the system controller. Check that there are no conflicts in the position of the various boards' memory addresses.

7. With the **d** and **m** boot ROM commands, verify that you can see the shared memory from the second processor. This is either the memory of the separate memory board if you are using the off-board configuration, or the dual-ported memory of the first processor if you are using the on-board configuration.

8. Using the **d** command on the second processor, look for the backplane anchor. The anchor begins with the telltale value of 0x1234. Also you should be able to see the backplane heartbeat – the second long word of the anchor, counting once per second.

9. When you have found the anchor from the second processor, enter the boot parameter for the boot device with that address as the anchor address:

```
boot device: bp=800000
```

   Enter the other boot parameters and try booting the second processor.

10. If the second processor does not boot, you can use *bpShow*( ) on the first processor to see if the second processor is attaching correctly to the backplane. If not, then you have probably specified the anchor address incorrectly on the second processor. If the second processor is attached, then the problem is more likely to be with the gateway or with the UNIX configuration.

11. You can use some of the UNIX tools to examine the state of the network from the UNIX side. Use **arp**, **netstat**, **etherfind**, and **ping** as suggested in **2.8 Troubleshooting.**

12. If all else fails, call your technical support organization.

7

Cross-Development

Contents

7

Cross-Development

7.1 Introduction

The Vx960 provides a highly integrated, cross-development environment. In Vx960, UNIX-based development systems support the development and maintenance of real-time applications that run on target systems under the Vx960 operating system. This chapter describes in detail the cross-development procedures used to create and run Vx960 systems and applications. It covers the following topics:

- The nature of the Vx960 cross-development environment.
- How to build, load, and run applications under Vx960.

Vx960 has been ported to numerous development and target systems, and can support many different hardware configurations. Some of the cross-development procedures discussed in this chapter depend somewhat on the exact system and configuration you are running. The procedures in this chapter are presented in generic form and may differ slightly on your particular system.

7.2 The Vx960 Cross-Development Environment

All Vx960 facilities are based on a single simple construct: the C subroutine (or C-compatible subroutine).

- All Vx960 facilities are provided by libraries of C subroutines.
- Any C subroutine can be invoked interactively, using the Vx960 shell.
- Any C subroutine can be spawned as a Vx960 task.
- Any C subroutine can be connected to an interrupt, a watchdog timer, or an auxiliary timer.

This uniformity makes Vx960 a very efficient development system. No special code must be written to interface to "system faults." No special processing is required to "build" a task. No special user interface programs or interactive test-bed programs must be written to test new code.

Instead, you simply write subroutines. As usual, of course, they can be called from other program modules to perform their function. But they can also be called interactively, from the shell, to perform their function spontaneously as needed, for testing or debugging. They can be spawned with their own task context, either from another program module or interactively from the shell. They can be connected to interrupts or timers, again either from another program module or interactively from the shell.

To provide this environment, Vx960 has a key set of facilities that set up the appropriate program environment to call subroutines in particular contexts: the *taskSpawn*( ) routine establishes the environment necessary for a routine to be executed as a task; and the routines *intConnect*( ), *sysAuxClkConnect*( ), and *wdStart*( ) establish the environment necessary for routines to be executed as the result of hardware interrupts, an auxiliary timer, and watchdog timer expirations, respectively.

Perhaps the most significant facility for the development environment is the Vx960 shell, from which you can interactively call any routine. This is made possible by the *system symbol table* maintained by the Vx960 module loader.

7.3 The Module Loader and System Symbol Table

Vx960 includes a module loader which loads object modules into memory from the host over a network or from a local disk. These object modules are in GNU/960 module format (*b.out* or *coff*; *b.out* is assumed in the following discussion), which includes the code, data, relocation directives, and a symbol table containing all of the module's subroutine and variable names and addresses.

When loading an object module, the Vx960 loader performs the following functions:

- Loads the module's code and data segments into memory.
- Relocates the module according to relocation directives in the object module.
- Resolves external references using the memory-resident, system-symbol table.
- Optionally, adds the module's symbols to the system-symbol table.

The system-symbol table thus contains the subroutine and variable names, and their addresses, from all loaded program modules, including Vx960 modules themselves. The system-symbol table then gives run-time access to subroutine and variable addresses by name.

As explained below, Vx960 makes heavy use of the system symbol table in two important areas: run-time linking and symbolic debugging under the Vx960 shell.

7.3.1 Run-Time Linking

The module loader uses the system symbol table to resolve undefined symbols in modules being loaded. This *run-time linking* allows applications to be built and maintained in conveniently sized pieces, rather than requiring the entire application to be linked together into a single module.

Furthermore, run-time linking makes it easy to have genuinely *shared* subroutine libraries, in which a single copy of a set of subroutines can be used by several tasks, rather than requiring each task to be linked with separate copies of needed subroutines.

However, there is the following limitation: the loader can only resolve names that have already been loaded in previous modules. Thus, modules must be loaded in order, according to their hierarchy of routine and variable references.

Of course, you can link some modules together on your host system, rather than have them linked by the Vx960 loader. This may be necessary if two modules cross-reference each other, prohibiting the hierarchical loading order mentioned above.

In some cases, it can also prove more convenient to link some modules together on the host, rather than loading each module separately under Vx960. Such linking of modules does not sacrifice their accessibility under Vx960 in any way, since the GNU/960 linker retains all the symbols of each module in the resulting composite symbol table.

It is important to understand that after a module has been downloaded, its symbols are not updated with subsequent downloads. The following example illustrates this point. Module A, which contains the routine *process_thing*(), is downloaded. Module B, which calls the routine *process_thing*(), is downloaded. The entry point to module B is the routine *makeItHappen*(). When module B is downloaded, its relocation-information is used to resolve the external reference to *process_thing*(). A bug is discovered and fixed in *process_thing*(), and a new module A is downloaded. It downloads correctly, but executing *makeItHappen*() still indicates the bug in *process_thing*(). This bug exists because *makeItHappen*() code still points to the old copy of *process_thing*(). If module B is downloaded again, it resolves *process_thing*() to the most recent downloaded copy, and *makeItHappen*() works correctly.

7.3.2 Symbolic Debugging

The other important use of the system symbol table is to provide interactive runtime access to routine and variable names. In particular, the Vx960 shell allows you to interactively call any subroutines and access any variables entered in the system symbol table.

This is particularly useful during symbolic and source-level debugging. For example, Vx960 offers symbolic disassembly of any loaded modules and symbolic traceback of the current nesting of subroutine calls of any task. You can also debug tasks from a source-level debugger running remotely on a UNIX machine.

7.4 Building and Loading Application Modules

In the Vx960 development environment, application modules for the target system are created and maintained on a UNIX host system. First, the source code, generally in C (or in a C-compatible language), is edited and compiled in the usual way to produce a relocatable object module. The resulting object modules can then be loaded and dynamically linked by a running Vx960 system via the network. The following sections describe these cross-development procedures in detail.

7.4.1 Compiling Application Modules

Vx960 can support multiple target architectures in a single development tree. To accommodate this, several Vx960 include files contain conditional compilation directives based on the definition of the variable CPU. When using these include files, the variable CPU must be defined in one of the following:

- the source modules
- the include files
- the compilation command line.

To define CPU in the source modules or include files, add the line:

```
#define CPU I960CA
```

To define CPU on the compilation command line, add the flag:

```
-DCPU=I960CA
```

With makefiles, the CPU definition can be added to the definition of the flags passed to the compiler (e.g., CFLAGS).

The GNU/960 tool set is used to compile application modules. The following is a typical command for compiling an application module for a Vx960 or compatible system:

```
% gcc960 -c -ACA -I/usr/vx/h -O3 applic.c
```

This command compiles the module **applic.c**. The meaning of the flags is as follows:

-c	specifies that the module is to be compiled only, and not linked for execution under UNIX. The output is an unlinked object module with the suffix **.o**, i.e., **applic.o**.
-I/usr/vx/h	gives access to the Vx960 include files (see the section below).
-O3	tells the compiler to perform various optimizations, in this case at level 3.
-ACA	tells the compiler to generate code for the 80960CA processor.

7.4.2 Using Vx960 Include Files

Many application modules make use of Vx960's operating system facilities or utility libraries. This may require that the source module include various Vx960 include files. The following sections discuss the use of Vx960 include files.

Vx960 supplies ANSI C function prototype declarations for all global routines in the Vx960 include files. The ANSI C prototypes are used by default with GNU/960; to suppress them, the macro __STDC__ must be undefined either in the application code before any include statements with the **#undef** C-preprocessor directive or on the command line using the **-traditional** flag.

Vx960 5.0 provides the ANSI header files **stddef.h**, **stdlib.h**, and **string.h**.

See the GNU/960 documentation for additional information on the compiler command line flags.

7.4.2.1 Vx960 Header File: vxWorks.h

The include file **/usr/vx/h/vxWorks.h** contains many basic definitions and types that are used extensively by other Vx960 modules. Many other Vx960 include files require these definitions, so this file should be included first by every application module that uses Vx960 facilities. The **vxWorks.h** file is included in a source module with the following line:

```
#include "vxWorks.h"
```

7.4.2.2 Other Vx960 Include Files

Application modules may include other Vx960 include files as needed to access Vx960 facilities. For example, an application module that used Vx960 linked-list subroutine library needs to include the lstLib.h file with the following line:

```
#include "lstLib.h"
```

The manual entry for each library lists all include files necessary to use that library.

7.4.2.3 UNIX Include Files

Application modules should not include certain UNIX include files for which there are Vx960 equivalents. The following table shows the correspondence between UNIX and Vx960 include files. Some of these include files resolve to GNU/960 include files.

Table 7-1. Vx960/UNIX Include File Equivalents

Vx960	UNIX
bout.h/coff.h	a.out.h
ctype.h	ctype.h
errno.h	errno.h
in.h	in.h
ioctl.h	ioctl.h
memLib.h	malloc.h
setjmp.h	setjmp.h
socket.h	socket.h
stdioLib.h	stdio.h
strLib.h	string.h
types.h	types.h

7.4.2.4 The -I Compiler Flag

By default, the compiler searches for include files first in the directory of the source module and then in the $G960BASE/include directory. The environment variable

G960BASE contains the path of GNU/960 installation directory. Refer to the appropriate GNU/960 documentation for additional information.

To access the Vx960 include files, the compiler must be directed to search /usr/vx/h for include files. This is done by specifying the flag:

```
-I/usr/vx/h
```

to the compiler.

7.4.2.5 Vx960 Nested Include Files

Some Vx960 facilities make use of other, lower-level Vx960 facilities. For example, the watchdog timer facility uses the linked-list subroutine library. The watchdog timer include file wdLib.h uses definitions that are supplied by the linked-list include file lstLib.h. Thus lstLib.h must be included before wdLib.h.

It would be undesirable to require that the application programmer know of such include file interdependencies and orderings. Instead, all Vx960 include files explicitly include any other prerequisite include files. Thus wdLib.h itself contains an include of lstLib.h. (The exception to this is the basic Vx960 include file vxWorks.h, which all other include files assume is already included.)

This, in turn leads to the problem that an include file could get included more than once, if one were included by several other include files, or was also included directly by the application module. Normally, including an include file more than once generates fatal compilation errors, because definitions get made more than once. To solve this problem, all Vx960 include files contain conditional compilation statements and definitions that ensure that their text only gets included once, no matter how many times they are specified by include statements.

In short, an application module can include just those include files it needs, without regard to interdependencies or ordering, and no conflicts will arise.

7.4.2.6 Network Include Files

In addition to the general include files in /usr/vx/h, the Vx960 network modules use a set of internal include files in /usr/vx/h/net. These include files should not be required by any application programs. Other than the Vx960 network modules themselves, the only modules that should include headers from /usr/vx/h/net are the network drivers for specific network controllers.

The directory /usr/vx/h also contains a subdirectory called rpc, which contains header files that must be included by applications using the remote procedure call library.

7.4.2.7 Hidden Parts of Vx960 Include Files

Vx960 modules are designed so that you never need to know or reference the modules' internal data structures. In general, all legitimate access to a facility is provided by the module's subroutine interfaces. The internal details should be thought of as "hidden" from the user. This means, in fact, that the internal implementations can change without affecting the users of the facilities.

Unfortunately, the C language provides no mechanisms to "export" only the "visible" portions of a module's interface, while keeping the internal details "hidden." As a compromise, however, Vx960 include files designate with comments those definitions and declarations that are supposed to be hidden. The hidden portions of include files are bracketed by comment lines of the form:

```
/* HIDDEN */
 ...
/* END_HIDDEN */
```

You should never make any references to any of the definitions in these hidden sections, nor should any assumptions be made based on those definitions. The only acceptable use of a module's facilities is through the visible definitions in the include file and the module's subroutine interfaces.

Although this rule is not currently enforced in any way, adherence ensures that your application code will not be affected by internal changes in the implementation of a Vx960 module.

7.4.3 Linking an Application Module

After an application module has been compiled, it can be loaded directly by the Vx960 *linking loader*. In general, application modules do not need to be linked with the GNU/960 linker, **gld960**. However, using **gld960** may be necessary if several application modules cross-reference each other such that the Vx960 loader cannot resolve all the external references as each is loaded. The following is a typical UNIX command to link several cross-referencing application modules:

```
% gld960 -o applic.o -r applic1.o applic2.o applic3.o
```

This creates another object module, **applic.o**, from the object modules **applic1.o**, **applic2.o**, and **applic3.o**. The **-r** flag causes the resulting object module to be left in relocatable form (i.e., not linked for standalone execution).

Vx960 facilities called by the application modules are left unresolved. These are resolved by the Vx960 loader when the module is loaded into Vx960 memory. Do not link each of the application modules with the Vx960 libraries. That would defeat the load-time, linking feature of Vx960, causing there to be multiple copies of Vx960 system modules.

Many of the Vx960 facilities come from the libraries supplied with the compiler. For example, a C run-time library supplied with your tool set might be used for *atoi( )*, *bcopy( )*, *sqrt( )*, etc. Similarly, floating point support can come from these libraries. However, these libraries are likely to include routines that conflict with or are incompatible with Vx960 facilities, so it is also important that you do not link your application code with these libraries directly if you wish to use the Vx960 runtime loader. Instead you must configure the Vx960 to include the facilities you need and let the Vx960 do the linking.

Just as Vx960 can be built with certain facilities disabled, Vx960 may not need all the routines in the libraries, so these routines may not be present in your Vx960 executable (*vxWorks*) and therefore cannot be resolved by the loader. You must ensure that your *vxWorks* image is built with the routines that you need. If these routines come from a source outside of Vx960, you must ensure that they do not conflict with the Vx960 facilities.

If you are using the GNU/960 tool set, you can define the configuration macro INCLUDE_GNU_960_LIBC in your *target*/config.h file to get those parts of **libcgca.a** (or **libcgka.a** or **libcgkb.a**) that are compatible with the Vx960 loaded with the rest of Vx960. Similarly, you can get the GNU/960 **libfpg.a** floating point library included by defining INCLUDE_GNU_960_LIBFP.

See 8.2 Configuring Vx960, 8.3 Building a Vx960 System Image, and the GNU/960 manuals for complete information.

7.4.4 Loading an Application Module

Once application object modules have been compiled (and possibly linked by gld960), they can be loaded by a running Vx960 system via the network. This involves invoking the Vx960 module loader from the Vx960 shell using the Vx960 command *ld*( ) (not to be confused with the UNIX ld command).

The following is a typical Vx960 load command from the Vx960 shell:

```
-> ld < host:/usr/fred/applic.o
```

This causes the object module /usr/fred/applic.o on the system called "host" to be loaded into memory, relocated, and linked to previously loaded modules. It is common to set the current default I/O device to be the host file system and directory containing your application modules, as follows:

```
-> cd "host:/usr/fred"
```

Then the load command becomes just:

```
-> ld < applic.o
```

Once application modules have been loaded into memory, any subroutine in the module can be invoked directly from the shell, spawned as a task, connected to an interrupt, and so forth.

8

Configuration

Contents

8

Configuration

8.1 Introduction

The Vx960 distribution files include a Vx960 system image for the target(s) ordered. A Vx960 *system image* is a binary module that can be booted and run on a target system. The system image consists of all the desired system object modules linked together into a single non-relocatable object module with no unresolved external references.

At first, you will find the supplied system image adequate for development. Eventually, you will want to tailor its configuration to your own needs. This chapter describes in detail the procedures used to configure and build Vx960 systems. It covers the following topics:

- Vx960 configuration files and configuration options and parameters
- How to build Vx960 system images with alternative configurations.

8.2 Configuring Vx960

Vx960 has many options and parameters that can be specified to tailor the system to particular hardware and software requirements. The configuration of Vx960 is almost completely determined by a single C language module, **usrConfig.c**, which contains the Vx960 start-up routines, and by the configuration headers **configAll.h** and **config.h**, which contain definitions of selected options and parameters.

The Vx960 distribution includes the configuration files for the default development configuration. Application builders can alter these files, or create their own versions, to better suit their particular configurations. Vx960 can then be rebuilt using the procedures described in the section below, **8.3 Building a Vx960 System Image**. This section describes the Vx960 configuration files in detail.

8.2.1 The Configuration Headers

Many of the most common Vx960 configuration options and parameters are controllable by definitions in the global configuration header **vx/config/all/configAll.h** and in the target-specific configuration header **vx/config/***target***/config.h**.

8.2.1.1 The Global Configuration Header: configAll.h

The **configAll.h** header, in the directory **vx/config/all**, contains default definitions that apply to all targets, unless redefined in the target-specific header **config.h**. The following options and parameters are defined in **configAll.h**:

- kernel configuration parameters
- I/O system parameters
- NFS parameters
- selection of optional software modules
- selection of optional device controllers
- device controller I/O addresses, interrupt vectors, and interrupt levels
- miscellaneous addresses and constants.

8.2.1.2 The Target-Specific Configuration Header: config.h

There is also a target-specific header, config.h, in the target-specific Vx960 directory **vx/config/*target***. This header contains definitions that apply just to that specific target. This header may also redefine default definitions in **configAll.h** that are inappropriate for the particular target. For example, if a particular target cannot access a device controller at the default I/O address defined in **configAll.h** because of addressing limitations, the I/O address may be redefined in **config.h**.

The **config.h** header includes definitions for the following parameters:

- default boot parameter string for boot ROMs
- interrupt vectors for system clock and parity errors
- target-specific device controller I/O addresses, interrupt vectors, interrupt levels
- backplane network parameters
- miscellaneous memory addresses and constants.

8.2.1.3 Selection of Optional Features

Vx960 includes optional features and device drivers that can be included or omitted from a Vx960 system. These are controlled by special symbols that can be defined or undefined in the configuration headers to cause conditional compilation in the **usrConfig.c** module. These symbols all start with the prefix INCLUDE_. Table 8-1 lists the optional elements and their control symbols.

For example, to include the network driver for the Intel 82596 LAN coprocessor, the symbol INCLUDE_EI should be defined, or to include the Network File System (NFS) facility, INCLUDE_NFS should be defined.

The distributed version of **configAll.h** includes all the available software options, and includes several network device drivers.

Exclusion of the shell and networking facilities is discussed in detail in **8.4 Alternative Vx960 Configurations**.

Table 8-1. Selectable Vx960 Options

Symbol	Option
Modules	
INCLUDE_ADA	Ada support
INCLUDE_DEBUG	Native debugging
INCLUDE_DOSFS	DOS-compatible file system
INCLUDE_ENV_VARS	UNIX compatible environment variables
INCLUDE_FLOATING_POINT	Floating point I/O
INCLUDE_FTPD	FTP server daemon
INCLUDE_NET_INIT	Network subsystem initialization
INCLUDE_NET_SHOW	Extended network status/information routines
INCLUDE_NETWORK	Network subsystem code
INCLUDE_NFS	Network File System (NFS)
INCLUDE_PIPES	Pipe driver
INCLUDE_RAMDRV	RAM disk driver
INCLUDE_RAWFS	"Raw" file system
INCLUDE_RDB	RDB Package (Remote Debugging)
INCLUDE_RLOGIN	Remote login with **rlogin**
INCLUDE_RPC	Remote Procedure Calls (RPC)
INCLUDE_RT11FS	RT-11 file system
INCLUDE_SECURITY	Remote login security package
INCLUDE_SHELL	C-expression interpreter shell
INCLUDE_SPY	Task activity monitor
INCLUDE_STDIO	Standard I/O package
INCLUDE_TELNET	Remote login with **telnet**
INCLUDE_TIMEX	Function execution timer
INCLUDE_SCSI	SCSI device support
INCLUDE_STAT_SYM_TBL	Create user-readable error status
INCLUDE_SYM_TBL	Symbol table package
INCLUDE_TASK_VARS	Task variable package

Table 8-1. Selectable Vx960 Options (continued)

Symbol	Option
Network Interfaces	
INCLUDE_BP	Backplane network interface
INCLUDE_EI	Intel 82596 LAN coprocessor Ethernet interface
INCLUDE_ENP	CMC Ethernet interface
INCLUDE_EX	Excelan Ethernet interface
INCLUDE_IE	SUN Ethernet interface
Other	
INCLUDE_NET_SYM_TBL	Load the system symbol table from the network.
INCLUDE_NFS_MOUNT_402	NFS mount the boot file system as in 4.0.2
INCLUDE_NFS_MOUNT_ALL	Mount all mountable NFS file systems from the host.
INCLUDE_STANDALONE_SYM_TBL	Use a built-in symbol table; do not load vxWorks.sym.
INCLUDE_STARTUP_SCRIPT_402	Run <host>:<bootdir>startup.cmd
INCLUDE_STARTUP_SCRIPT	Execute a startup script after booting.

8.2.2 The Configuration Module: usrConfig.c

Many Vx960 options and parameters can be set simply by modifying definitions in the configuration headers described in the previous section. However, for some applications it may be desirable to configure Vx960 in ways not provided by the configuration headers, and in such cases, application builders can modify the code in the configuration module usrConfig.c.

The usrConfig.c module consists of four main pieces:

usrInit( ) an initialization routine that runs before multitasking is enabled,

usrRoot( ) the initial task to be executed,

usrClock( ) the routine that is called on every interrupt of the system clock,

usrNetInit( ) the routine that initializes the network (called by *usrRoot( )*).

The elements of usrConfig.c are best understood by going through each step of the system start-up sequence. The next sections trace the flow of control during the booting and initialization of a Vx960 system.

8.2.3 The Initial Entry Point: *sysInit( )*

To bring up a Vx960 system, get an image of a Vx960 system into main memory. This is usually done via the Vx960 boot ROM. Once a Vx960 system image has been loaded into the proper position in main memory, the boot ROM transfers control to the Vx960 start-up entry point, *sysInit( )*.

This entry point is in the system-dependent, assembly-language module, sysALib.s. It locks out all interrupts, sets up the initial C stack pointer, and jumps to *usrInit( )*, a C subroutine in the usrConfig.c module. For some targets, *sysInit( )* also performs some minimal system-dependent hardware initialization, just enough to execute the remaining initialization in *usrInit( )*.

8.2.4 The Initial Routine: *usrInit( )*

The *usrInit( )* routine, in the usrConfig.c module, first does all the initialization that must be performed before the multitasking kernel is actually started, and then starts the kernel execution. It is the first C code to run in Vx960. It is invoked in supervisor mode with all hardware interrupts locked out.

There are many Vx960 facilities that cannot be invoked from this routine. In general, since there is no task context as yet (e.g., no TCB and no task stack), facilities that require a task context cannot be invoked. This includes any facility that might cause the caller to be suspended, such as semaphores or any facility that uses them. Instead, the *usrInit( )* routine does only what is necessary to create an initial task, *usrRoot( )*, that will complete the start-up.

The initialization in *usrInit( )* includes the following:

- Zeroing out the system *bss* segment: The C language specifies that all uninitialized variables will have initial values of 0. These uninitialized variables are put together in a segment called *bss*. This segment is not actually loaded during the

bootstrap, since it is known to be all 0. Since the *usrInit( )* routine is the first C code to be executed, it clears the section of memory containing *bss* as its very first action.

- **Initializing interrupt vectors:** Before enabling interrupts and starting the kernel, the interrupt and fault vectors of the i960 microprocessor must be set up. The routine *excVecInit( )* is called to initialize all exception vectors to default handlers that will report exceptions caused by program errors or unexpected hardware interrupts.
- **Initializing system hardware to a quiescent state:** System hardware is initialized by calling the system dependent routine *sysHwInit( )*. This mainly consists of resetting and disabling hardware devices that may cause interrupts once interrupts are enabled when the kernel is started. This is important because the Vx960 interrupt handlers for I/O devices, system clocks, etc, are not connected to their interrupt vectors until the system initialization is completed in the *usrRoot( )* task.
- **Initializing the multitasking kernel:** The last act of *usrInit( )* is to initialize the multitasking kernel by calling the routine *kernelInit( )*. This call specifies:
  - the entry point and stack size of the root task
  - start and end address of the initial memory pool
  - the interrupt stack size and lock level.

  The call actually initiates the multitasking environment and never returns. The *usrInit( )* thread of execution is terminated and control is transferred to the root task, *usrRoot( )*.

8.2.4.1 Initializing the Memory Pool

Vx960 includes a memory allocation facility, in the module memLib, that manages a pool of available memory. The *malloc( )* routine allows callers to obtain variable size blocks of memory from the pool. Internally, Vx960 uses *malloc( )* for all dynamic allocation of memory. In particular, many Vx960 facilities allocate data structures upon initialization. Therefore, the memory pool must be initialized before any other Vx960 facilities are initialized.

Vx960 makes heavy use of *malloc( )*, including allocation of space for loaded modules, allocation of stacks for spawned tasks, and allocation of data structures upon initialization. Application programmers are also encouraged to use *malloc( )* to allocate memory as needed. Therefore, it is recommended that you assign to the Vx960

memory pool all unused memory, unless you specifically must reserve some fixed absolute memory area for a particular application use.

The memory pool is initialized in the routine *kernelInit( )*. The parameters to *kernelInit( )* specify the start and end address of the initial memory pool. In the default *usrInit( )* distributed with Vx960, the pool is set to start immediately following the end of the booted system, and to contain all the rest of available memory.

The extent of available memory is determined by the routine *sysMemTop( )* which is a system-dependent routine that determines the size of available memory by probing specific addresses. If your system has other non- contiguous memory areas that you would like to make available in the general memory pool, you can add them to the pool by calling the routine *memAddToPool( )* later in the *usrRoot( )* task.

8.2.5 The Initial Task: *usrRoot( )*

Once the multitasking kernel is executing and control is transferred to the *usrRoot( )* task, all Vx960 multitasking facilities are available, and the initialization of the system can be completed. The standard *usrRoot( )* task distributed with Vx960 performs the following:

- initialization of the system clock
- initialization of the I/O system and drivers
- initialization of the network
- initialization of optional facilities
- creation and loading of the system symbol table
- execution of a user-supplied startup script
- spawning of the shell.

You are free to modify these initializations to suit your particular configuration. The meaning of each step and the significance of the various parameters are explained in the following sections.

8.2.5.1 Initialization of the System Clock

The first action in the *usrRoot( )* task is to initialize the Vx960 clock. The system clock interrupt vector is connected to the system clock service routine *usrClock( )* (described later), by calling the routine *sysClkConnect( )*. Then, the system clock rate is set to 60 ticks per second by calling the routine *sysClkRateSet( )*.

8.2.5.2 Initialization of the I/O System

The Vx960 I/O system is initialized by calling the routine *iosInit( )*. The arguments specify the maximum number of drivers that may be subsequently installed, the maximum number of files that may be open in the system simultaneously, and the desired name of the "null" device which is built into the Vx960 I/O system. This null device is a "bit-bucket" on output and always returns end-of-file for input.

8.2.5.3 Creation of the Console Devices

Next, the driver for the primary or console serial ports is installed by calling the driver's initialization routine, which is typically called *tyCoDrv( )*. The actual devices are then created and named by calling the driver's device creation routine, typically *tyCoDevCreate( )*. The arguments to this routine include the device name, a channel number if the driver supports multiple ports, and input and output buffer size. The macro NUM_TTY, specifying the number of tty ports (default 2), and the macro CONSOLE_TTY, specifying which port is console (default 0), are specified in configAll.h but may be overridden in config.h for boards with a non-standard number of ports.

8.2.5.4 Setting of Standard In and Standard Out

Next, establish the system-wide standard in, standard out, and standard error assignments by opening the desired devices and calling *ioGlobalStdSet( )*. These are used throughout Vx960 as the default device for communicating with the user. The devices are set to have the options usual for interactive terminals by calling *ioctl( )* to set the device options to OPT_TERMINAL.

8.2.5.5 Installation of Exception Handling, Debugging, and Logging

It is desirable to initialize the Vx960 exception handling facilities (supplied by the module excLib), debugging facilities (supplied by dbgLib), and logging facilities (supplied by logLib), as soon as possible in the root task. This facilitates detection and debugging of program errors in the root task itself or in the initialization of the various facilities.

The exception handling facilities are initialized by calling *excInit( )*. This will spawn the exception support task, *excTask( )*. Following this initialization, program errors

causing hardware faults are safely reported, and hardware interrupts to uninitialized vectors are reported and dismissed. The Vx960 signal facility, used for task-specific exception handling, is initialized by calling *sigInit*( ).

The logging facilities are initialized by calling *logInit*( ). The arguments specify the fd of the device to which logging messages are to be written, and the number of log message buffers to allocate. The logging initialization also includes spawning the logging task, *logTask*( ).

The debugging facilities are initialized, if they have been selected by defining INCLUDE_DEBUG, by calling *dbgInit*( ).

8.2.5.6 Initialization of Standard I/O

Vx960 includes an optional *standard I/O* package that provides the functionality of the UNIX buffered I/O facility. This can be included by defining the symbol INCLUDE_STDIO in the header configAll.h.

8.2.5.7 Creation of File System Devices

Many Vx960 configurations include at least one disk device or RAM disk with a DOS, RT-11, or "raw" file system. Before any disk drivers or the RAM-disk driver are initialized, the file system module must be initialized by calling the routine *dosFsInit*( ), *rt11FsInit*( ), or *rawFsInit*( ). The argument specifies the maximum number of file descriptors per file system that may be open simultaneously.

Next, any disk drivers are installed by calling the drivers' initialization routines. The actual devices can then be created and named by calling the driver's device creation routine. The arguments to these routines depend on the particular driver, but typically include the device name, a drive number if the driver supports multiple drives, and possibly some media specifications (single/double density/sided, etc.).

8.2.5.8 Initialization of Floating Point Support

Support for floating point I/O is initialized by calling the routine *floatInit*( ). This facility is included by defining the symbol INCLUDE_FLOATING_POINT in configAll.h.

8.2.5.9 Inclusion of Performance Monitoring Tools

Vx960 has two performance monitoring tools. A task activity summary is provided by **spyLib**, and a subroutine execution timer is provided by **timexLib**. These facilities are included by defining the symbols **INCLUDE_SPY** and **INCLUDE_TIMEX**, respectively, in **configAll.h**.

8.2.5.10 Initialization of the Network

Before the network can be used, it must be initialized. This is handled by the routine *usrNetInit( )*, which is called by *usrRoot( )*. The routine *usrNetInit( )* takes a configuration string as an argument. This configuration string is usually the "boot line" that was specified to the Vx960 boot ROMs to boot the system. Based on this configuration string, *usrNetInit( )* does the following network initialization:

- Initializes the network subsystem by calling the routine *netLibInit( )*.
- Attaches and configures the appropriate network drivers.
- Adds gateway routes.
- Initializes the remote file access driver **netDrv** and adds the remote file access device.
- Initializes the remote login facilities.
- Optionally initializes the remote procedure calls (RPC) facility.
- Optionally initializes the remote source debugging facility.
- Optionally initializes the Network File System (NFS) facility. As noted above, the inclusion of some of these network facilities are controlled by definitions in the **configAll.h** header file.

The network initializations are described in detail in **6. Network**.

8.2.5.11 Creation and Loading of the System Symbol Table

The Vx960 interactive development environment relies heavily on its memory resident *system symbol table*, which contains the names and addresses of variables and functions that have been loaded into memory. This symbol table is generally obtained from the host system from which Vx960 was booted. However, in "standalone" configurations (described below, under **8.4 Alternative Vx960 Configurations**)

the system symbol table is not read in; in such configurations, the remainder of this section does not apply.

First, the system symbol table is created by calling *symTblCreate*( ). For convenience during debugging, it is most useful to have access to all symbols in the system. On the other hand, a production version of a system can be built that does not require any system symbol table at all, if memory use is a factor.

The call to *symTblCreate*( ) creates an empty system symbol table. Vx960 system facilities are not accessible interactively via the shell until the symbol definitions for the booted Vx960 system are entered into the system symbol table. This is done by reading the system's symbol table from a file called **vxWorks.sym** in the same directory from which **vxWorks** was loaded. This file contains an object module that consists only of a symbol table section containing the symbol definitions for all the variables and functions in the booted system module. It has zero-length (empty) code, data, and relocation sections. Nonetheless, it is a legitimate object module in the standard object module format.

The symbols in **vxWorks.sym** are entered in the system symbol table by calling *loadSymTbl*( ) (whose source is in **usrConfig.c**). This routine uses the Vx960 loader to load symbols from **vxWorks.sym** into the system symbol table.

8.2.5.12 Execution of a Startup Script

Before finally turning control over to the shell, *usrRoot*( ) executes a user-supplied startup script if this option has been enabled. The startup script is executed if its filename was specified at boot time with the startup-script parameter (see **2.5 Booting Vx960**). If the parameter is missing, no startup script is executed. This feature is enabled by defining INCLUDE_STARTUP_SCRIPT in **configAll.h**.

Startup scripts can be particularly useful for tailoring the configuration of a Vx960 system without having to rebuild it. For more information on the use of startup scripts, see **9.4.3 Scripts.**

8.2.5.13 Spawning of the Shell

In a typical development configuration, the *usrRoot*( ) task spawns the Vx960 shell task by calling *shellInit*( ). The first argument to *shellInit*( ) specifies the shell's stack size which must be large enough to accommodate any routines you call from the shell. The second argument is a boolean that specifies whether the shell's input is

from an interactive source (TRUE), or a non-interactive source (FALSE) such as a script file. If the source is interactive, then shell prompts for commands but does not echo them to standard out, and vice-versa if the source is non-interactive.

8.2.6 The System Clock Routine: *usrClock( )*

The final element in the usrConfig.c module is the system clock interrupt routine *usrClock( )*. This interrupt-level routine is attached to the system clock timer interrupt by the *usrRoot( )* task described above. The routine must call the kernel clock tick routine *tickAnnounce( )*. Application developers may also add any application-specific processing to this routine.

8.3 Building a Vx960 System Image

The configuration of Vx960 is determined by the source files discussed in the previous section, **8.2 Configuring Vx960**. If any of these files are modified to alter the configuration, Vx960 must be rebuilt. This includes re-compiling certain modules and re-linking the system image. This section explains the procedures for rebuilding the Vx960 system image.

8.3.1 Rebuilding Vx960 with make

Vx960 uses the UNIX make facility to re-compile and re-link modules. The file **Makefile** is provided in each Vx960 target directory and contains the directives for rebuilding Vx960 for that target. (See the UNIX manual entry for **make** for more information on makefiles.)

To rebuild Vx960, change to the Vx960 target directory for the desired target, and invoke **make** as follows:

```
% cd /usr/vx/config/target
% make
```

make re-compiles and re-links modules as necessary, based on the directives in the target directory's Makefile.

8.3.2 Linking the System Modules

The UNIX commands to link a Vx960 system image are somewhat complicated. Fortunately, it is not necessary to understand those commands in detail since the Makefile in each Vx960 target directory includes the necessary commands. However, for completeness, this section gives an explanation of the flags and parameters used by Vx960.

Most of the Vx960 operating system modules are distributed in the form of UNIX archive libraries including **all.a, net.a, config.a,** etc. There is a complete set of these libraries in vx/lib/*arch* for each architecture supported by Vx960.

These modules are combined with the configuration module usrConfig.o by the GNU/960 linker gld960. (The usrConfig.c module is described in the previous section, **8.2 Configuring Vx960.**) The following is a typical command for linking a Vx960 system:

```
gld960 -o vxWorks -ACA -T 1000 -e _sysInit \
    sysALib.o sysLib.o tyCoDrv.o usrConfig.o version.o \
    /usr/vx/lib/i960ca/all.a /usr/vx/lib/i960ca/config.a \
    /usr/vx/lib/i960ca/net.a /usr/vx/lib/i960ca/rpc.a \
    /usr/vx/lib/i960ca/netif.a /usr/vx/lib/i960ca/wind.a
    /usr/vx/lib/i960ca/i960.a -lcg -lfpg
```

The meanings of the flags in this command are as follows:

-o vxWorks — This causes the resulting object module to be named **vxWorks.**

-ACA — This flag specifies the use of the 80960CA-specific version of the C runtime library.

-T 1000 — This flag specifies the relocation address, in this example 1000 hex. This is the address at which the system must be loaded in the target, and is also the address at which execution starts. Some target systems may have limitations on where this relocation address can be.

-e _sysInit — This flag defines the entry point to vxWorks. Note that *sysInit*( ) is the first routine in sysALib.o, which is the first module loaded by ld.

sysALib.o and sysLib.o
These are modules with CPU-dependent initialization and support routines. The module **sysALib.o** must be the first module specified in the **ld** command.

tyCoDrv.o This is the CPU-specific console I/O driver.

usrConfig.o This is the configuration module described in detail in a section below. If you have several different system configurations, you may have several different configuration modules, either in /usr/vx or in your own directory.

version.o This module contains data structures which define the creation date and version number of this vxWorks object module. It is created by compiling the output of **makeVersion**.

/usr/vx/lib/i960ca/all.a
This is the archive format library that contains all the system-independent Vx960 modules. This library is searched, and the necessary Vx960 modules included, to satisfy the external references of the **usrConfig** module.

/usr/vx/lib/i960ca/config.a
This is the archive format library that contains all the I/O drivers and user-alterable Vx960 modules. This library is searched and the necessary Vx960 modules included, to satisfy external references.

/usr/vx/lib/i960ca/net.a
This archive contains all of the Vx960 network support code.

/usr/vx/lib/i960ca/rpc.a
This archive contains the Remote Procedure Call library.

/usr/vx/lib/i960ca/netif.a
This archive contains the network interface drivers; for example, **if_ei.c** for the Intel 82596 coprocessor and ENP-10/L Ethernet controllers.

/usr/vx/lib/i960ca/wind.a
This archive contains the Vx960 kernel.

/usr/vx/lib/i960ca/i960.a

This archive contains the processor specific, optimized assembly routines.

-lcg This is the GNU/960 C run-time library. This library is searched and necessary routines included, to satisfy remaining unresolved references. Typically, this includes internal compiler support routines. The **-lcg** must come after the Vx960 libraries. Otherwise, some GNU/960 routines are linked in instead of Vx960 routines with identical names (the I/O library, for instance).

-lfpg This is the GNU/960 C floating-point support library. This library is searched and necessary routines included, to satisfy remaining unresolved references. Typically, this includes internal compiler support routines. The **-lfpg** must come after the Vx960 libraries.

Note: Refer to Section **7.4.3 Linking an Application Module** to allow optimum code sharing of routines from the C run-time library and the C floating-point library.

Additional object modules:

Additional object modules (with **.o** suffix) can be linked in with the Vx960 system simply by naming them on the **gld960** command. Note that during development, application object modules are generally not linked with the system unless they are needed by the **usrConfig** module since they may be incrementally loaded after Vx960 is booted. See Section **8.4 Alternative Vx960 Configurations** for more detail on linking application modules in a bootable system.

8.3.3 Creating the System Symbol Table Module

In a standard Vx960 configuration, part of the initialization procedure is for Vx960 to read its own symbol table, so system symbols are available to applications loaded later, and from the shell. The symbol table file is created by a the supplied tool **xsym**. Processing an object module with **xsym** creates a new object module that contains all the symbols of the original file, but with no code or data. The line in **Makefile** that creates this file is:

```
xsym < vxWorks > vxWorks.sym
```

The file **vxWorks.sym** is the symbol table loaded during Vx960 initialization.

8.4 Alternative Vx960 Configurations

The preceding discussion of the **usrConfig** module outlined a typical configuration for a development environment. In this configuration, the Vx960 system image contains all of the Vx960 modules that are necessary to allow you to interact with the system through the shell. Once the system is up, you can interactively load, execute, and debug other Vx960 and application modules, via the Vx960 shell.

However, as you approach a debugged "production" version of your application, you may want to add application modules to the bootable system image and add application startup to the system initialization. In a final version, you may also want to remove the shell altogether, create a stand-alone application requiring no network, or create a system completely bootable from ROMs.

The following sections discuss these alternatives to the typical development configuration.

8.4.1 Excluding Development Facilities

In a production configuration, it may be desirable to remove some of Vx960 development facilities to reduce the memory requirements of the system, to reduce boot time, or for security purposes.

Optional Vx960 facilities can easily be omitted from Vx960 by commenting out or using *#undef* to undefine their corresponding control constants in the header files **configAll.h** or **config.h**. For example, debugging facilities can be omitted by undefining INCLUDE_DEBUG. RT-11 file support and the RAM-disk driver can be omitted by undefining INCLUDE_RT11FS and INCLUDE_RAMDRV, respectively. Many more facilities can be omitted in this way. See the list of INCLUDE options in Table 8-1.

8.4.1.1 Excluding the Shell

Application startup can be made a part of system initialization, in which case there may be no need to interact with the system through the Vx960 shell. The shell can be excluded by undefining INCLUDE_SHELL. If the shell is excluded, there is no need to look up symbol names; thus, the creation and loading of the system symbol table can also be omitted. This is done by undefining INCLUDE_NET_SYM_TBL.

If the shell is excluded, *usrRoot( )* spawns a simple demo which serves as an example for initializing bootable applications. This routine loops forever prompting for a string and echoing it. If the string "0" or "1" is entered, it displays various memory statistics. For more information, see **8.4.2 Creating Bootable Applications**.

8.4.1.2 Excluding Network Facilities

In some applications you might want to eliminate the Vx960 network facilities. For example in the ROM-based systems or stand-alone configurations described below, there may be no need for the network facilities.

To exclude the network facilities, undefine the options INCLUDE_NETWORK, INCLUDE_NET_INIT, and INCLUDE_NET_SYM_TBL. To exclude the Remote Procedure Call library (RPC) undefine INCLUDE_RPC, INCLUDE_NFS, and INCLUDE_RDB.

Any configuration option that requires another configuration option automatically includes that option. For example, if the INCLUDE_RPC is undefined, but INCLUDE_NFS is defined, NFS will be included for RPC. NFS requires RPC to function. The file /usr/vw/config/configAll.h details these interdependencies.

8.4.2 Creating Bootable Applications

In general, during development it is preferable not to include extraneous modules in the Vx960 bootable system, since any changes to those modules would then require recreating the Vx960 system. However, as you approach a final version of your system, you may want to add application modules to the bootable system image, and include startup of your application with the system initialization routines. In this way, you can create a "bootable" application, which is completely initialized and functional after booting, without requiring any interaction with the Vx960 shell.

To include your application modules in the bootable system image, add the names of the application object modules (with the .o suffix) to the list of modules to be linked with the gld960 command. Look for the Makefile entry *vxWorks* and add your object modules after the $(LD) instruction.

To start up your application with the initialization of the system, add the necessary code to the *usrRoot*( ) routine in usrConfig.c. This code can call application initialization routines, create additional I/O devices, spawn application tasks, and so forth, just as you might do from the shell in a development configuration. The following model for this is provided: since users typically exclude the shell from bootable applications, usrConfig.c has been set up to include and initialize a simple demo if INCLUDE_SHELL is not defined – *usrRoot*( ) spawns *usrDemo*( ) as a task instead of the shell. If you are excluding the shell, replace the spawning of *usrDemo*( ) with the appropriate initialization of your application. For additional information on the demo and removing the shell, see **8.4.1.1 Excluding the Shell.**

Application size is an important consideration in building bootable applications. Vx960 boot ROM code is copied to starting address 0x90000 in RAM, and the system image is copied to 0x1000 on Vx960 systems. If the entire system image requires 570 Kbytes or more, it writes over the boot ROM code while downloading, thereby killing the booting process. To prevent this, reprogram the boot ROMs to copy the boot ROM code to a sufficiently higher memory address by increasing the value of RAM_HIGH_ADR in the makefile.

Then remake the boot ROMs by typing:

```
% make bootrom.hex
```

The binary image size of typical boot ROM code is 256 Kbytes or less.

8.4.3 Creating a Stand-Alone Vx960 System with a Built-In Symbol Table

It is sometimes necessary to create a Vx960 system that does not load its own symbol table during initialization. This might be necessary, for instance, if network facilities are not initialized or included and the system does not have access to a device from which it can read the symbol table.

The procedure for building such a system is somewhat different from the normal procedure described above. No change is necessary in the application code or in usrConfig.c. Only the make procedure must be changed.

In the target directory's Makefile, one of the Vx960 versions that can be built is vxWorks.st, which includes a built-in symbol table. The following is a copy of a typical Makefile entry for creating vxWorks.st:

```
vxWorks.st : usrConfig_st.o $(MACH_DEP) $(LIBS) $(LDDEPS)
        @ $(RM) $@ symTbl.c symTbl.o
        @ make version.o
        $(LD) -o tmp.o -r $(MACH_DEP) usrConfig_st.o version.o $(LIBS)
        $(BIN)makeSymTbl tmp.o > symTbl.c
        $(CC) $(CFLAGS) -S symTbl.c
        sed -e 's/_\(.*\)_DOT_EL_EF_/\1.lf/g' \
             -e 's/_DOT_EL_EF_/.lf/g' symTbl.s > tmp.s
        $(AS) -o symTbl.o tmp.s
        $(LD) -o $@ $(LD_LOW_FLAGS) -e _sysInit tmp.o symTbl.o
        @ $(RM) tmp.o tmp.s tmp.c version.o
```

The module usrConfig_st.o is the usrConfig.c module compiled with the STANDALONE flag defined. The STANDALONE flag causes the usrConfig.c module to be compiled with the built-in system symbol table.

The STANDALONE flag also suppresses the initialization of the network, since the network is not now needed to download the system symbol table. If you want the network initialized, either call *usrNetInit*( ) from the shell, or modify *usrRoot*( ) to call *usrNetInit*( ).[1]

The makefile macro references are defined as follows:

$(MACH_DEP) — the machine-dependent modules sysALib.o, sysLib.o and tyCoDrv.o.

1. While vxWorks.st suppresses network initialization, it still includes the network. Specifically, the STANDALONE flag defines INCLUDE_STANDALONE_SYM_TBL and INCLUDE_NETWORK and undefines INCLUDE_NET_SYM_TBL and INCLUDE_NET_INIT.

$(LIBS)	the archives in /usr/vx/lib.
$(LD) and $(CC)	architecture-specific commands for loading and compiling.
$(BIN)	architecture-specific /usr/vx/bin directory.

Vx960 is linked as described previously, except that the first pass through the loader does not specify the final load address, so that the output from this stage is still relocatable.

The **makeSymTbl** tool is invoked on the loader output; it constructs a data structure containing all the symbols in Vx960. This structure is then compiled and linked with Vx960 itself to produce the final bootable Vx960 object module.

To include your own application in the system image, add the object modules after the first **$(LD)** instruction and observe the procedures discussed in the previous section.

To create the **vxWorks.st** version, type:

```
% make vxWorks.st
```

8.4.4 Creating a Vx960 System In ROM

8.4.4.1 General Procedures

To put Vx960 or a Vx960-based application into ROM, modify the command to the UNIX loader to load the module **romInit.o** before **sysALib.o**. The entry point option must then be changed to **-e _romInit**. The **romInit** routine initializes the stack to point directly below the text segment. It then calls *bootInit*( ), which clears memory and copies the Vx960 text and data segments to the proper location in RAM. Control is then passed to *usrInit*( ).

A good example of a ROM-based Vx960 application is the Vx960 boot ROM program itself. The file **/usr/vx/config/all/bootConfig.c** is the configuration module for the boot ROM, replacing the file **usrConfig.c** provided for the default Vx960 development system. The makefiles in the target-specific directories contain directives for building the boot ROMs, including an Intel Hex format file suitable for downloading to a PROM programmer. This is done by invoking the UNIX make command as follows:

```
% make bootrom.hex
```

Vx960 target makefiles also include the entry **vxWorks.st_rom** for creating a ROMable version of the stand-alone system described in the previous section. **vxWorks.st_rom** differs from **vxWorks.st** in two respects: (1) **romInit** code is loaded as discussed above, and (2) the portion of the system image that is not essential for booting is compressed by approximately 40% using the Vx960 **compress** tool (see below). To use this makefile entry and to create an Intel Hex version, type:

```
% make vxWorks.st_rom.hex
```

When adding application modules to a ROMable system, size is an important consideration. Keep in mind that by using the **compress** tool, what normally requires a 256-Kbyte ROM may well fit into a 128-Kbyte ROM. Also, space can be gained by removing the shell and system symbol table, which account for approximately 66% of the standard Vx960 configuration.

If the ROMs' contents are to exceed 256 Kbytes, modify both config.h and the makefile to reset ROM_SIZE.

By default, **romInit** copies the boot ROM contents into memory starting at address 0x90000. However, target boards with only 1 Mbyte of RAM are unable to accommodate more than 448 Kbytes at this address. This can be remedied by resetting ROM_TEXT_ADRS in the makefile to copy the ROM code to a sufficiently lower memory address. Since no system image is downloaded over the network as in a standard configuration, the normal entry point at 0x1000 can be used for this.

8.4.4.2 Boot ROM Compression

Vx960 boot ROMs are compressed to about 60% of their actual size using a binary compression algorithm, which is supplied as the tool **compress**. When control is passed to the ROMs on system reset or reboot, a small (8 Kbytes) uncompress routine, which is *not* compressed, is executed. It then uncompresses the remainder of the ROM into RAM and jumps to the start of the uncompressed image in RAM. There is a short delay during the uncompression (about 4 seconds) before the Vx960 prompt appears.

This mechanism is also available to application builders who wish to compress a ROMable Vx960 application. The Makefile entry for vxWorks.st_rom demonstrates how this can be accomplished. For more information, see also the manual entries for **bootInit**(1) and **compress**(3).

9

Shell

Contents

9

Shell

9.1 Introduction

Much of Vx960's power comes from the uniformity of the environment it offers the application developer. A key element of this environment is the interactive access to system and application facilities provided by the Vx960 *shell*.

Most systems are supplied with some kind of command interpreter program which provides the user with a limited and fixed set of commands that can be invoked interactively. The Vx960 shell, however, provides a much simpler and more powerful mechanism: the shell allows you to interactively invoke any subroutine that has been loaded into memory, including both those supplied in Vx960 modules and those in application modules.

In fact, the shell can interactively interpret almost any C language expression, including execution of most C operators and resolution of symbolic data references and subroutine invocations.

The shell is not required in a Vx960 system. It can be omitted, in a production system for example, by suppressing its spawning in the root task.

9.2 General Use

9.2.1 Accessing the Shell

The shell can be accessed in one of two ways: via direct serial connection, or via the network. When Vx960 is first booted, the shell's terminal is the system console. The UNIX tools **rlogin** or **telnet** are used to access the shell via the network. Network access is discussed in **9.8 Remote Login**.

9.2.2 Terminal Control Characters

The table below lists special terminal characters frequently used for shell control:

Table 9-1. Vx960 Shell Special Characters

Command	Description
^H	Delete a character (backspace).
^U	Delete an entire line.
^C	Abort and restart the shell.
^X	Reboot (fault to the ROM monitor).
^S	Temporarily suspend output.
^Q	Resume output.
ESC	Toggle between input mode and edit mode.

The first four of these are defaults that can be mapped to different keys using routines in tyLib(1); see also **4.6.2.3 Tty Special Characters**.

For more information on the use of these characters, see **9.5 Shell Line Editing** and **9.6 Aborting the Shell**.

9.2.3 Shell Examples

The Vx960 shell reads lines of input from an input stream, parses and evaluates each line, and writes the result of the evaluation to an output stream. The shell accepts the same expression syntax as the C compiler with a few variations.

This simple mechanism can be used in many different ways. The following examples illustrate some typical uses of the shell. Note that the shell prompt for interactive input is "->". User input is shown in **bold face** and shell responses are shown in regular face.

- **Data Conversion:**

```
-> 68
value = 68 = 0x44 = 'D'

-> 0xf5de
value = 62942 = 0xf5de = _init + 0x52

-> 's'
value = 115 = 0x73 = 's'
```

  The shell prints all integers and characters in both decimal and hexadecimal, and if possible, as a character constant or a symbolic address and offset.

- **Data Calculation:**

```
-> (14 * 9) / 3
value = 42 = 0x2a = '*'

-> (0x1355 << 3) & 0x0f0f
value = 2568 = 0xa08

-> 4.3 * 5
value = 21.5
```

  Almost all C operators can be used.

- **Calculations With Variables:**

```
-> (j + k) * 3
value = ...

-> *(j + 8 * k)
(...address (j + 8 * k)...): value = ...

-> x = (val1 - val2) / val3
new symbol "x" added to symbol table
```

```
address = ...
value = ...

-> f = 1.41 * 2
new symbol "f" added to symbol table
f = (...address of f...): value = 2.82
```

Variable *f* will get an 8-byte floating point value.

- **Invocations of Vx960 Subroutines:**

```
-> errnoGet ( )
value = (...value returned...)

-> taskSpawn ("tmyTask", 10, 0, 1000, myTask, fd1, 300)
value = ...

-> fd = open ("file", 0)
new symbol "fd" added to symbol table
fd = (...address of fd...): value = ...

-> fprintf (fd, "status = %d.", status)
value = ...
```

- **Invocations of Application Functions:**

```
-> testFunc (123)
value = ...

-> myValue = myFunc (1, &val, testFunc (123))
myValue = (...address of myValue...): value = ...

-> myFloat = (float ()) myFuncWhichReturnsAFloat (x)
myFloat = (...address of myFloat...): value = ...
```

For cases in which a routine does not return a 4-byte integer, see the section 9.3.3.4 Function Calls below.

9.2.4 The Nature of the Shell: A Caveat

The Vx960 shell serves a dual role. First, it acts as a command interpreter that provides access to all Vx960 facilities, by allowing the user to call any Vx960 subroutine.

Second, as can be seen from the examples above, the shell can be used as a powerful debugging tool for the application developer. Application modules can be tested and debugged interactively by calling any application routine.

However, the power of the shell makes it a potentially dangerous facility.You must know the consequences of calling a particular routine, and you must specify correctly the arguments to be passed to it. By invoking routines incorrectly, it is possible to cause system failures that require rebooting the system.

Thus the shell is not an appropriate interface for naive end-users. But it is an excellent interface for application developers.

9.3 Shell Language

9.3.1 Data Types

The most significant difference between the Vx960 shell and the C compiler lies in the way that they handle data types. The shell does not accept any C declaration statements, and no data-type information is available in the system symbol table. Instead, an expression's type is determined by the types of its terms.

Unless explicit type-casting is done, the shell makes the following assumptions about data types:

- In an assignment statement, the type of the left hand side is determined by the type of the right hand side.
- If floating-point numbers and integers both appear in an arithmetic expression, the resulting type is a floating-point number.
- Data types are assigned to various elements as shown in Table 9-2.

A constant, variable, or return value of a routine can be treated as a different type than what the shell assumes it to be by specifying the type in a fashion similar to standard C type-casting. Table 9-3 shows the various data types available in Vx960, along with examples of how they can be set and referenced.

Table 9-2. Vx960 Shell Data-Type Assumptions

Element	Data Type
variable	*int*
variable used as floating-point	*double*
return value of subroutine	*int*
constant with no decimal point	*int/long*
constant with decimal point	*double*

Table 9-3. Vx960 Data Types

Type	Bytes	Set Variable	Display Variable
int	4	`x = 99`	`x` `(int) x`
long	4	`x = 33` `x = (long)33`	`x` `(long) x`
short	2	`x = (short)20`	`(short) x`
char	1	`x = 'A'` `x = (char)65` `x = (char)0x41`	`(char) x`
double	8	`x = 11.2` `x = (double)11.2`	`(double) x` `printf ("%g", (double)x)` `printf ("%f", (double)x)` `printf ("%e", (double)x)`
float	4	`x = (float)5.42`	`(float) x` `printf ("%g", (float)x)` `printf ("%f", (float)x)` `printf ("%e", (float)x)`

Strings, or character arrays, are not actual data types in the Vx960 shell. To declare a string, set a variable to a string value. For example:

```
-> ss = "shoe bee doo"
```

The variable *ss* is a pointer to the string "shoe bee doo". To display *ss*, type:

```
-> printf ("%s", ss)
```

or:

```
-> d ss
```

The *d( )* command displays memory where *ss* is pointing. The shell places no type restrictions on the application of operators. For example, the shell expression:

```
*(1000 + 3 * 16)
```

evaluates to the 4-byte integer value contained in memory location 1048.

9.3.2 Lines and Statements

The shell parses and evaluates its input a line at a time. A line may consist of a single shell statement or several shell statements separated by semicolons. A semicolon is not required on a line containing a single statement. A statement cannot continue on more than one line.

Shell statements are either C expressions or assignment statements.

9.3.3 Expressions

Shell expressions consist of literals, symbolic data references, function calls, and the usual C operators.

9.3.3.1 Literals

The shell interprets the literals in the table below in the same way as the C compiler, with one addition: the shell also allows hex numbers to be preceded by $ instead of 0x.

Table 9-4. Vx960 Shell Literals

Literal	Example
decimal numbers	`143967`
octal numbers	`017734`
hex numbers	`0xf3ba` or `$f3ba`
floating point numbers	`666.666`
character constants	`'x'` and `'$'`
string constants	`"hello world!!!"`

9.3.3.2 Data Variable References

Shell expressions may contain references to data variables whose names have been entered in the system symbol table. Unless a particular type has been specified for a data variable, the variable's value in an expression is the 4-byte value at the memory address obtained from the symbol table. It is an error if an identifier in an expression is not found in the symbol table, except in the case of assignment statements discussed below.

The C compiler prefixes all user-defined identifiers with an underline, so that the identifier *myVar* is in the symbol table as *_myVar*. The identifier can be entered either way to the shell – the shell searches the symbol table for a match either with or without a prefixed underline.

Shell expressions can also access data in memory that does not have a symbolic name in the symbol table but whose address is known to the user. This can be done with the C indirection operator * applied to a constant. For example, *0x1000 refers to the 4-byte integer value at memory address 1000 hex.

9.3.3.3 Operators

The shell interprets the following operators in the same way as the C compiler:

Table 9-5. Vx960 Shell Operators

Operator Type	Operators					
arithmetic	+	-	*	/	unary -	
relational	==	!=	<	>	<=	>=
shift	<<	>>				
logical	||	&&	!			
bitwise	|	&	~	^		
address and indirection	&	*				

The shell assigns the same precedence to the operators as the C compiler. However, unlike the C compiler, the shell always evaluates both sub-expressions of the logical binary operators || and &&.

9.3.3.4 Function Calls

Shell expressions may contain calls to C functions (or C compatible functions) whose names have been entered in the system symbol table.

The shell executes such function calls as though the function had been called, with the specified arguments, from the body of the shell itself, in the context of the shell task. The value of a function call is the 4-byte integer value returned by the function.

The shell allows up to ten arguments to be passed to a function. In fact, the shell always passes ten arguments to every function called, passing values of zero for those arguments not specified. This is harmless since the UNIX C function-call protocol handles passing of variable numbers of arguments. However, it allows trailing arguments of value zero to be omitted from function calls in shell expressions.

Function calls can be nested. That is, a function call can be an argument to another function call.

Shell expressions can also contain references to function addresses instead of function invocations. As in C, this is indicated by the absence of parentheses after the function name. Thus the expression:

```
myFunc ( ) + 4
```

evaluates to the result returned by the function *myFunc*( ), added to 4, while the expression:

```
myFunc + 4
```

evaluates to the address of *myFunc*( ) added to 4. However, an important exception to this occurs when the function name is the very first item encountered in a statement. This is discussed in the section below.

Shell expressions can also contain calls to functions that do not have a symbolic name in the symbol table, but whose addresses are known to the user. This can be done by supplying the address in place of the function name. Thus the expression:

```
0x1000 ()
```

would call a parameterless function whose entry point is at address 1000 hex.

9.3.3.5 Omitting Parentheses

In practice, most statements input to the shell are function calls, often to invoke Vx960 facilities. To simplify this use of the shell, an important exception is allowed to the standard expression syntax required by C. When a function name is the very first item encountered in a shell statement, the parentheses surrounding the function's arguments can be omitted. Thus the following shell statements are synonymous:

```
-> rename ("oldname", "newname")

-> rename "oldname", "newname"
```

as are:

```
-> errnoGet ( )

-> errnoGet
```

9.3.3.6 Task References

Most Vx960 routines that take a task parameter require a task ID. However, when invoking routines interactively, specifying a task ID can be cumbersome since the ID is an arbitrary and possibly lengthy number.

To accommodate interactive use, Vx960 shell expressions can reference a task by either task ID or task name. The Vx960 shell attempts to resolve a task argument to a task ID; if no match is found in the system symbol table, it searches for the argument in the list of active tasks. When it finds a match, it substitutes the task name with its matching task ID. In symbol lookup, symbol names take precedence over task names.

By convention, task names are prefixed with a "t". This avoids name conflicts with entries in the system symbol table. The names of system tasks and the default task names assigned when tasks are spawned use this convention. For example, tasks spawned by *sp( )* are given names such as *t1*, *t2*, and *t3*. Users are encouraged to adopt this convention.

9.3.4 Assignments

The Vx960 shell interprets assignment statements in the form:

addressExpression = *expression*

The left side of an expression must evaluate to an addressable entity, i.e., a legal C value. The data type of the left side is determined by the type of the right side. If the right side does not contain any floating point constants or non-integer type-casts, then the type of the left side will be an integer. The value of the right side of the assignment is put at the address indicated by the left side. For example, the assignment:

```
-> x = 0x1000
```

sets the 4-byte integer variable *x* to 0x1000. The assignment:

```
-> *0x1000 = x
```

sets the 4-byte integer value at memory address 0x1000 to the current value of *x*.

The compound assignment:

```
-> x += 300
```

adds 300 to the 4-byte integer variable *x*. The assignment:

```
-> *0x1000 += 300
```

adds 300 to the 4-byte integer value at memory address 0x1000. The compound assignment operator **-=**, as well as the auto-increment and auto-decrement operators **++** and **--**, are also available.

9.3.4.1 Automatic Creation of New Variables

New variables can be created automatically by assigning a value to an unknown identifier (one not found in the system symbol table) with an assignment statement.

When the shell encounters such an assignment, it allocates space for the variable and enters the new identifier in the system symbol table along with the address of the allocated variable. The new variable is set to the value and type of the right-side expression of the assignment statement. The shell prints a message indicating that a new variable has been allocated and assigned the specified value.

For example, if the identifier *fd* does not exist in the system symbol table, the statement:

```
-> fd = open ("file", 0)
```

creates a new variable named *fd* and assign to it the result of the function call.

9.3.5 Comments

The shell allows two kinds of comments. First, comments of the form /* ... */ can be included anywhere on a shell input line. These comments are discarded, and the rest of the input line evaluated as usual. Second, any line whose first non-blank character is # is ignored. Comments are useful for Vx960 shell scripts. See the section **9.4.3 Scripts** below.

9.3.6 Strings

When the shell encounters a string literal ("...") in an expression, it allocates space for the string including the null byte string terminator. The value of the literal is the address of the string in the allocated storage. For instance, the expression:

```
-> x = "hello there"
```

allocates 12 bytes from the memory pool, enters the string in those 12 bytes (including the null terminator), and assigns the address of the string to *x*. The expression:

```
-> free (x)
```

can be used to put those 12 bytes back into the memory pool. See **memLib**(1) for information on memory management.

Furthermore, even when a string literal is not assigned to a symbol, memory is still allocated for it, e.g.:

```
-> printf ("hello there")
```

uses 12 bytes of memory that are never freed. This is because if strings were temporarily allocated, and a string literal were passed to a routine being spawned as a task, then by the time the task executes and accesses the string, the shell would have released, and possibly even reused, the temporary storage where the string was held.

9.3.7 Ambiguity of Arrays and Pointers

In a C expression, a non-subscripted reference to an array has a special meaning, namely the address of the first element of the array. The shell, to be compatible, should use the address obtained from the symbol table as the value of such a reference, rather than the contents of memory at that address. Unfortunately, the information that the identifier is an array as all data type information, is not available after compilation. For example, if a module contains the following:

```
char string [ ] = "hello";
```

one might be tempted to enter a shell expression like:

```
-> printf (string)                                    0
```

While this would be correct in C, the Vx960 shell passes the first 4 bytes of the string itself to *printf( )*, instead of the address of the string. To correct this, the shell expression must take the address of the identifier:

```
-> printf (&string)                                    ❷
```

To make matters worse, if the identifier had been declared a character pointer instead of a character array:

```
char *string = "hello";
```

then ❶ would be correct and ❷ would be wrong! This is confusing since C allows pointers to be subscripted like arrays, such that the value of *string[0]* would be "h" in either of the above declarations.

The moral of the story is that in shell expressions, unlike in C itself, array references and pointer references are different. In particular, array references require an explicit application of the address operator &.

9.3.8 Pointer Arithmetic

While C language treats pointer arithmetic specially, Vx960 does not, since it treats all non-type-cast variables as 4-byte integers.

In the Vx960 shell, pointer arithmetic is no different than integer arithmetic. It is valid, but it does not take into account the size of the variable pointed to. Consider this example:

```
-> *(myPtr + 4) = 5
```

Assume that the value of *myPtr* is 0x1000. In C, if *myPtr* is a pointer to a type *char*, this would put the value 5 in the byte at address at 0x1004. If *myPtr* is a pointer to a 4-byte integer, the 4-byte value 0x00000005 would go into bytes 0x1010–0x1013. The Vx960 shell, on the other hand, treats variables as integers, and therefore would put the 4-byte value 0x00000005 in bytes 0x1004–0x1007.

9.4 Redirection

The shell and many other facilities in Vx960 direct I/O to three known streams, known as standard input, standard output, and standard error. These streams are set and accessed through the subroutine library **fioLib**. See **4. I/O System** and the manual entry for **fioLib**(1).

The shell provides a *redirection* mechanism for momentarily reassigning the standard input and standard output streams for the duration of the parse and evaluation of an input line. The redirection streams are indicated by < and > symbols followed by stream names, at the very end of an input line. No other syntactic elements can follow the redirection specifications. The redirections are in effect during the evaluation of all statements on the line.

For example, the input line:

```
-> copy < input > output
```

would cause the standard input stream of **fioLib** to be set to the stream named *input* and the standard output to the stream named *output* during the execution of *copy*( ).

If the file to which standard output is redirected does not exist, it is created on the default device.

9.4.1 Ambiguity Between Redirection and Operators

Ambiguity exists between redirection specifications and the relational operators *less than* and *greater than*. The shell always assumes that an ambiguous use of < or > specifies a redirection rather than a relational operation. Thus the ambiguous input line:

```
-> x > y
```

would cause the printout of the value of the variable *x* to be written to the stream named *y*, rather than comparing the value of variable *x* to the value of variable *y*. However, a semicolon can always be used to remove the ambiguity, since the shell requires that the redirection specification be the last element on a line. Thus the input lines:

```
-> x; > y

-> x > y;
```

are unambiguous. The first line prints the value of the variable *x* to the output stream *y*. The second line prints on standard output the value of the expression "*x* greater than *y*."

9.4.2 The Nature of Redirection

The redirection mechanism of the Vx960 shell is different from that of the UNIX shell, although the syntax and terminology are similar.

Vx960 has standard input and output streams for each task. In Vx960, standard input and standard output are a single pair of available open streams, commonly used by system and application modules for debugging and diagnostic I/O.

For example, you might be tempted to specify redirection streams when spawning a routine as a task, intending to send the output of *printf*( ) calls in the new task to an output stream, while leaving the shell's I/O directed at the current terminal. This is not possible. For example, the shell input line:

```
-> taskSpawn (...myFunc...) > output
```

momentarily redirects the global standard output during the brief execution of the spawn routine, but does not affect the I/O of the resulting task.

9.4.3 Scripts

9.4.3.1 Redirecting Shell I/O

A special case of I/O redirection concerns the I/O of the shell itself; that is, redirection of the streams the shell's input is read from, and its output is written to. The syn-

tax for this is the usual redirection specification on a line containing no other expressions. This causes the shell to call itself, recursively, after redirecting the I/O.

Thus the new invocation of the shell does I/O to and from the redirected streams. New invocations of the shell return to the original shell after reaching an end-of-file on their input streams. This returns the standard input and output to their previous streams, and resumes the processing of those streams by the original shell.

The typical use of this mechanism is to have the shell to read and execute lines from a file on a disk or across the network. For example, the input lines:

```
-> <startup                                    ❶

-> <mars:/usr/fred/startup                     ❷
```

would cause the shell to read and execute the commands in the file **startup**, either on the default device as in ❶ or explicitly on the networked machine **mars** as in ❷. If one had already changed to mars:/usr/fred using *cd*( ), commands ❶ and ❷ would be equivalent.

Such command files are called *scripts*. Scripts are processed like input from an interactive terminal. After reaching the end of the script file, the shell returns to processing I/O from the original streams.

An easy way to create a shell script is from a list of commands you have executed in the shell. The history routine *h*( ) prints a list of the last 20 commands typed to the shell. Thus, typing:

```
-> h > host:/tmp/newscript
```

creates a file **host:/tmp/newscript** which contains the current shell history.

Scripts can also be nested. That is, scripts can contain shell input redirections that cause additional invocations of the shell to process other scripts.

9.4.3.2 System Startup Scripts

Vx960 shell scripts can be useful for setting up your environment at system startup. After Vx960 is booted, and before the shell task is spawned, a user-specified startup script is executed if its file name is supplied with the "s=" boot parameter (see **2.5 Booting Vx960**). If the parameter is missing, no startup script is executed. This feature is enabled by defining INCLUDE_STARTUP_SCRIPT in configAll.h.

Like the **.login** file of the UNIX C-shell, startup scripts can be used for setting system parameters to personal preferences, e.g., defining variables, resetting backspace or abort keys, and so forth. They can also be useful for tailoring the configuration of your system without having to rebuild Vx960, e.g.:

- Creating additional devices
- Loading and starting up application modules
- Adding a complete set of network host names and routes
- Setting NFS parameters and mounting NFS partitions.

9.5 Shell Line Editing

The shell provides a history mechanism similar to the UNIX K-shell history facility with a built-in line editor that allows you to edit previously typed commands.

The ESC key switches the shell from normal input mode to *edit mode*. The **vi**-like commands shown in the table **Vx960 Shell Line-Editing Commands** are used in edit mode. Note that some line-editing commands switch the editor to insert mode until an ESC is typed (as in **vi**) or a RETURN gives the line to the shell. RETURN always gives the line to the shell from either input or edit mode.

As mentioned earlier, the shell history command *h*( ) (in input mode) displays up to 20 of the previous commands typed to the shell; old commands fall off the top as new ones are entered. To edit a command, type ESC followed by one of the search and editing commands listed in the table.

See the manual section on the line-editing library **ledLib**(1) for more information.

Note: Since the shell toggles between *raw mode* and *line mode*, type-ahead can be lost. See **4.6.2.2 Raw Mode and Line Mode** for more information.

Vx960 Shell Line-Editing Commands

Command	Action
	Basic Control
ESC	- Switch to line-editing mode from regular input mode.
RETURN	- Give line to shell and leave edit mode.
^U	- Delete line and leave edit mode.
^L	- Redraw line.
^D	- Complete symbol name.
	Movement and Search
*n*G	- Go to command number *n*.
/*s* or ?*s*	- Search for string *s* backward in history, or forward.
n	- Repeat last search.
*n*k or *n*-	- Get *n*th previous shell command.
*n*j or *n*+	- Get *n*th next shell command.
*n*h or ^H	- Go left *n* characters.
*n*l or SPACE	- Go right *n* characters.
*n*w or *n*W	- Go *n* words forward, or *n* blank-separated words.
*n*e or *n*E	- Go to end of the *n*th next word, or blank-separated word.
*n*b or *n*B	- Go back *n* words, or *n* blank-separated words.
$	- Go to end of line.
0 or ^	- Go to beginning of line, or first non-blank character.
f*c* or F*c*	- Find character *c*, searching forward, or backward.

Command	Action
	Insert and Change
a or A	...ESC - Append, or append at end of line.
i or I	...ESC - Insert, or insert at beginning of line.
c SPACE	...ESC - Change character.
cl	...ESC - Change character.
cw	...ESC - Change word.
cc or S	...ESC - Change entire line.
c$ or C	...ESC - Change from cursor to end of line.
c0	...ESC - Change from cursor to beginning of line.
R	...ESC - Type over characters.
*n*r*c*	- Replace the following *n* characters with *c*.
~	- Toggle case, lower to upper or vice versa.
	Delete
*n*x	- Delete *n* characters starting at cursor.
*n*X	- Delete *n* character to left of cursor.
d SPACE or dl	- Delete character.
dw	- Delete word.
dd	- Delete entire line.
d$ or D	- Delete from cursor to end of line.
d0	- Delete from cursor to beginning of line.
	Put and Undo
p or P	- Put last deletion after cursor, or before cursor.
u	- Undo last command.

Note: The default value for *n* is 1.
The ESC which terminates insertion commands is unnecessary if the line is given to the shell with RETURN.

9.6 Aborting the Shell

You might want to abort the shell's evaluation of a statement. For example, an invoked routine may loop excessively, suspend, or wait on a semaphore. This can happen as the result of errors in arguments specified in the invocation, errors in the implementation of the routine itself, or oversight as to the consequences of calling the routine at all.

In such cases it is possible to abort and restart the shell task. This is done by pressing the special abort character on the keyboard, by default ^C. This causes the shell task to restart execution at its original entry point.

When restarted, the shell reassigns the system standard input and output streams to the original assignments they had when the shell was first spawned. Thus any shell redirections are canceled, and any executing shell scripts are aborted.

The abort facility works if:

- *dbgInit*( ) has been called (see **10. Debugging**).
- *excTask*( ) is running.
- The driver for the particular keyboard device supports it (all Vx960 supplied drivers do).
- The device's abort option is enabled. This is done with an *ioctl*( ) call, usually in the root task in **usrConfig.c**. For information on enabling the shell abort character, see **4.6.2.1 Tty Options.**

The abort key can be changed to a character other than ^C by calling *tyAbortSet*( ).

Also, you can enter an expression which causes the shell to incur a fatal error such as a bus error. Such an error results in the suspension of the offending task, which allows further debugging.

However, when such an error is incurred by the shell task, Vx960 restarts the shell, since further debugging would be impossible without access to it. Note that for this reason, as well as to allow the use of breakpoints and single-stepping, when debugging a routine it is often useful to spawn it as a task instead of calling it from the shell (see **10. Debugging** for more information).

When the shell is aborted for any reason, either because of a fatal error or because it is aborted from the terminal, a task trace is displayed. This trace shows where the shell was executing when it died.

An offending routine can have left things in a state which cannot be cleared when the shell is aborted. For instance, it may have taken a semaphore, which cannot be given as part of the abort.

9.7 The User Library

Although all Vx960 routines are available whenever needed, Vx960 includes some libraries of routines that are geared towards simple interactive use.

The library **usrLib** includes routines for:

- reporting system and task information
- loading object modules from files
- spawning, deleting, holding, and resuming tasks
- displaying and modifying memory
- examining the system symbol table
- listing system devices and disk directories
- copying and renaming files
- other higher-level facilities.

The **usrLib** library also includes the routine *help( )* that prints a list of commonly used routines and their arguments. See the manual entry **usrLib**(1) for details.

Other Vx960 modules have sets of routines designed to be called from the shell, including **dbgLib**, **rlogLib**, **timexLib**, and **spyLib**. Many other modules have status display routines that are intended for interactive use.

Application developers are encouraged to add their own interactive routines, either to **usrLib** or to separate modules. Note that the interactive use of routines is enhanced when they do additional parameter verification, error checking, error reporting, and so forth.

9.7.1 Help Routines

Several libraries include their own help routines. To find them all, type:

```
-> lkup "Help"
```

For more information, see the manual entry for *lkup*( ) in usrLib(1).

9.8 Remote Login

9.8.1 Remote Login from UNIX: rlogin and telnet

When Vx960 is first booted, the shell's terminal is normally the system console. To access the shell from a UNIX host via the network, type (on UNIX):

```
% rlogin targetname
```

where *targetname* is the name of the target Vx960 system in the UNIX file /etc/hosts. If **rlogin** is not available on your host system, use **telnet**.

A message is printed on the Vx960 system console indicating that the shell is being accessed via **rlogin** or **telnet**, and that it is no longer available from its console.

If the shell is being accessed remotely, typing at the console terminal has no effect. Vx960 is a single-user system – it allows access to the shell only from the console or from a single remote login session. To prevent someone from remotely logging in while you are at the console, use the routine *shellLock*( ) by typing:

```
-> shellLock 1
```

To make the shell available again, type:

```
-> shellLock 0
```

Since remote login can be accessed via a UNIX window on a workstation-type environment, it is possible to access more than one target system via **rlogin** at a time from the same UNIX workstation.

To end a remote-login shell session, type:

```
-> logout
```

or, if you are using rlogin:

```
-> ~.
```

(TILDE, DOT) as the only characters on a line.

9.8.2 Remote Login Security

You can be prompted to enter a login user name and password when accessing Vx960 remotely:

```
Vx960 login:      user_name
Password:         password
```

The remote-login security feature is enabled by defining INCLUDE_SECURITY in the configuration file configAll.h and rebuilding Vx960. The default login user name and password provided with the supplied system image is "target" and "password". You can change the user name and password by using the routine *loginUserAdd( )* from the shell. For example:

```
loginUserAdd "fred", "encrypted_password"
```

To obtain *encrypted_password*, use the tool **vxencrypt** on the UNIX host. It prompts you to enter your password, and then display the encrypted version.

To define a group of login names, a list of *loginUserAdd( )* commands can be included in a startup script and executed after the system has been booted. For more information on the use of startup scripts, see 9.4.3 Scripts.

A group of login names can be defined as part of the Vx960 system configuration by adding a list of *loginUserAdd( )* commands to the file usrConfig.c and rebuilding Vx960.

The remote-login security feature can be disabled at boot time by specifying the flag bit 0x20 (SYSFLAG_NO_SECURITY) in the *flags* parameter on the boot line. This feature can also be disabled by undefining INCLUDE_SECURITY in the configuration file configAll.h and rebuilding Vx960.

9.8.3 Remote Login from Vx960: *rlogin( )*

Using *rlogin( )*, you can access the UNIX host or another Vx960 system on the network from Vx960. From the Vx960 shell, type:

```
-> rlogin " sysname "
```

You then have access to the Vx960 or UNIX shell on the machine called *sysname*. When finished, use **logout** or *logout( )* to return to the original Vx960 shell.

9.9 The Shell Task

The Vx960 shell is spawned as a task by the root task in the module **usrConfig**. There are other tasks associated with the shell including **rlogin** and **telnet** tasks, if those facilities are included in Vx960.

The shell and other tasks associated with it are all created with the **VX_UNBREAKABLE** option; therefore breakpoints cannot be set in those tasks. This is necessary, since a breakpoint in the shell (or in one of the **rlogin** tasks during an **rlogin**) would make it impossible for the user to interact with the system. See the manual entry **dbgLib**(1) for information on "unbreakable" tasks.

Only one shell can run on a Vx960 system at a time; the shell's parser is not currently reentrant since it is implemented using the UNIX tool **yacc**.

10

Debugging

Contents

10

Debugging

10.1 Introduction

Vx960 provides numerous facilities for debugging application code. This chapter describes these debugging facilities and discusses general techniques for debugging under Vx960.

When debugging, you will be interacting with the Vx960 shell and with several libraries available through it. Therefore, the first step is to understand the shell. This chapter includes a short synopsis of how the shell fits into the debugging environment; however, for more detail see **9. Shell.**

The shell utilities for debugging are simply C subroutines designed to be called as if they were commands, although they can also be called from any other program. Vx960 debugging tools provide the following facilities:

- task-specific breakpoints
- task-specific single-stepping
- symbolic disassembler
- task and system information utilities
- symbolic task trace-back utility
- ability to call user routines
- ability to create and examine variables symbolically
- ability to examine and modify memory
- fault handling.

Manual entries for most of the utilities described in this chapter will be found in usrLib(1), excLib(1), or dbgLib(1).

10.1.1 On-Line Help

An on-line help facility is available as a "memory jogger." Two commands to get help are:

help	Display help for user utilities.
dbgHelp	Display help for debugging utilities.

You can get a list of all the help routines available by typing:

```
-> lkup "Help"
```

10.1.2 Source-Language Debugging

Vx960 target processes can also be debugged using the gdb-based debugger, a C source-language debugger running on the UNIX host.This debugger is an extended version of the GNU Source-Level Debugger (GDB) that allows full source-level debugging of remote Vx960 applications from a variety of host UNIX systems.

10.2 The Shell

Since the Vx960 shell is an integral part of Vx960's debugging facilities, this section reviews the basics of shell usage.

The Vx960 shell provides a powerful facility: the ability to evaluate almost any C expression interactively. This includes the ability to use data variables and functions that are contained in the system symbol table. Any command you type is interpreted as a C expression. The shell evaluates that expression and, optionally, assigns the result to a data variable.

Some examples:

```
-> 68                                                    ❶
value = 68 = 0x44 = 'D'

-> (25 * 3) & 0x48                                       ❷
value = 72 = 0x48 = 'H'

-> myVar                                                 ❸
value = 115 = 0x73 = 's'

-> myVar = func (12)                                     ❹
value = ...

-> newVar = 100                                          ❺
new symbol newVar added to symbol table
address = ...
value = 100 = 0x64 = 'd'
```

These examples perform the following:

❶❷ These simply cause expressions to be evaluated and the results displayed.

❸ looks up the value of *myVar* (which must be found in the system symbol table) and displays the value contained there.

❹ calls the routine *func*( ) with a single parameter (12), assigns the value returned to *myVar*, and displays the value.

❺ creates a new variable *newVar*, since it did not already exist, and assigns it the value 100.

10.2.1 Subroutines as "Commands"

One way of using the shell's expression handling capability is with simple expressions such as the following:

```
-> func (x, y)

-> func x,y
```

These two lines are equivalent; the parentheses are optional. In either case, *func*( ) is called with two parameters, and the resulting value is displayed on the console.

In fact, Vx960 provides many routines which are meant to be called from the shell interactively. These routines can be thought of as *commands*, rather than as *subroutines*, even though they are, in fact, normal C subroutines. All the commands discussed in this chapter fall in this category. When you see the word *command*, you can read *subroutine*, or vice versa, since their meaning here is identical.

10.2.2 Arguments to Commands

Many of the commands described here accept one or more arguments. These arguments, once again, can be arbitrary C expressions. Some examples:

```
-> d (1000)                    ❶
-> d 1000                      ❷
-> d 0x1000                    ❸
-> d dog                       ❹
-> d &dog                      ❺
-> d dog + 100                 ❻
-> d func (dog)                ❼
-> d func                      ❽
```

The command *d( )* will display a block of memory starting at the address which is passed to it as a parameter. The examples perform the following:

❶❷ display starting at address 1000 decimal.

❸ displays starting at 1000 hex.

❹ displays starting at the address contained in the variable *dog*.

❺ displays starting at the address of *dog*. (For example, if *dog* is a data variable at location 0x1234, and that memory location contains the value 10000, *d( )* would display starting at 10000 in example ❹ and at 0x1234 in ❺.)

❻ displays starting at 10100.

❼ calls the routine *func( )* with the parameter 10000 and then calls *d( )*, passing it the value returned by *func( )*.

❽ will display the code of *func( )* as a simple hex memory dump.

Notice that, just as in C, symbols that refer to code (*text* in UNIX jargon) evaluate to their address if no parentheses are given, unlike data and *bss* variables, which evaluate to their contents.

Many of the commands described here take optional arguments. The shell always passes exactly 10 arguments to any function it calls. If there are not that many typed on the command line, any arguments not typed are passed as 0. If none are typed, all 10 arguments are 0. When commands have optional arguments, some default is used when the command receives 0 as an argument, which normally means that you did not type an argument to the shell. For example:

```
-> l func, 20                                    ❶
-> l func                                        ❷
-> l                                             ❸
```

The *l*( ) command disassembles memory. (The *l*( ) command is described later in detail – see **10.6 Disassembler.**) In the examples:

❶ will disassemble 20 instructions in memory starting at the beginning of the routine *func*( ).

❷ uses a default count, and will disassemble however many instructions were specified on the previous *l*( ) command (the default begins at a reasonable value, if you never give a count).

❸ uses a default address as well. In this case, it will disassemble starting at the instruction following the previous *l*( ) command, or starting at the next instruction to be executed by a task doing single-stepping, or starting at the last breakpoint or exception fault that was hit. It will use whichever of these is most recent.

For example:

```
breakpoint in task t30 at 0x1234 = myFunc + 0x30
-> l                                             ❶
-> l                                             ❷
-> l func                                        ❸
```

Task *t30* has hit a breakpoint. In the above:

❶ will disassemble starting at 0x1234 until (let's say) 0x1250.

❷ will begin at the following instruction (perhaps 0x1254).

❸ will disassemble starting at the beginning of *func*( ).

10.2.3 Aborting the Shell

Occasionally you may find it necessary to abort the shell. For example, if you call a routine that is not fully debugged, or with the wrong parameters, or that waits on a semaphore that is not forthcoming, you may hang forever. In this case, abort the shell using ^C (since the called routine is operating in the shell's task context).

When you abort the shell, a task trace (described in detail below) is displayed so that you can determine where you were when the abort key was pressed.

10.3 Task Names and IDs, and the "Current" Task and Address

When a task is spawned, the user specifies a *task name* which is an ASCII string of arbitrary length. The system returns a *task ID* which is an efficient 4-byte pointer to the task's data structures. Task IDs are displayed with the *i*( ) command.

Most Vx960 low-level task functions that take a task parameter require a task ID. However, when invoking routines interactively, specifying a task ID can be cumbersome since the ID is an arbitrary and possibly lengthy number.

To accommodate interactive use, Vx960 shell expressions can reference a task by either task ID or task name. The Vx960 shell attempts to resolve a task argument to a task ID; if no match is found in the system symbol table, it searches for the argument in the list of active tasks. When it finds a match, it substitutes the task name with its matching task ID. In symbol lookup, symbol names take precedence over task names.

By convention, task names are prefixed with a "t". This avoids name conflicts with entries in the system symbol table. The names of system tasks and the default task names assigned when tasks are spawned use this convention. For example, tasks spawned by *sp*( ) will be given names such as *t1*, *t2*, and *t3*.

Many commands (*c( )*, *cret( )*, *s( )*, *so( )*, *ti( )*, *tt( )*) have a task parameter which may be omitted. If omitted, the *current* task is used. There is also a current address that *l( )* and *d( )* use if no address is specified. The current task and address are set when:

- A task hits a breakpoint or an exception fault. The current address is the address of the instruction that caused the break or exception.
- A task is single-stepped. The current address is the address of the *next* instruction to be executed.
- Any of the commands that use the current task or address are executed with a specific task parameter. The current address will be the address of the byte *following* the last byte that was displayed or disassembled.

10.4 Task and System Information

The following commands are available to get information about the state of the system and individual tasks:

`i` — Information. This command gives a snapshot of what tasks are in the system, and some information about each of them, such as state, PC, SP, and TCB address.

`ti` [*task*] — Task information. This command gives all the information contained in a task's TCB. This includes everything shown for that task by an *i( )* command, plus all the task's registers, and the links in the TCB chain. If *task* is 0 (or nothing), the current task is reported on.

`tt` [*task*] — Task trace. This command traces a task's stack, shows what routine the task is currently executing, and how it got there.

`checkStack` [*task*] — Check stack usage. This command displays a summary of a task's stack usage, or of all tasks if *task* is omitted. The summary includes the total stack size (SIZE), the current number of stack bytes used (CUR), the maximum number of stack bytes used (HIGH), and the number of bytes never used at the top of the stack (MARGIN = SIZE - HIGH). This routine is useful for

determining how much stack space to allocate, and for detecting stack overflow.

10.4.1 Task Trace: *tt( )*

Task trace shows the calling sequence that brought a task to its current state. For example:

```
_> tt tTelnetd
  65fb0 _vxTaskEntry    +10 : _telnetd ()
  2e5d0 _telnetd        +c0 : _accept ()
  29684 _accept         +a4 : _ksleep ()
  4e89c _ksleep         +3c : _semTake (0, 0, 1ce750, 344d8, 0, )
value = 0 = 0x0
```

The telnet daemon is in the routine *semTake* which was called from the kernel routine *ksleep*, which was called from the accept call in *telnetd*.

Task trace does have some limitations. It can only give routine names if they are in the system symbol table. It expects all routines to be C-compatible.

Routine parameters are only displayed for the first and last routines on the stack. The parameters of the first routine are saved in a known location on the stack at task creation so that a task may be restarted with *taskRestart( )*. The i960 calling sequence passes the parameters of a routine in the global registers. This routine may save these parameters in local registers, on the stack, or may modify them without saving at all. As a result, none of the parameters of the rest of the routines on the stack are in known locations at the time the task trace is done. For the last routine on the stack, *tt( )* reports the contents of the first several global registers as they are at the time of the *tt( )* call. This report may or may not accurately reflect the parameters to the routine. The parameters are only guaranteed to be accurate if the task in question is stopped at the beginning of the last routine on the stack. (For more information on calling sequences, see your tool-set documentation.)

The *tt( )* command can also sometimes display the wrong number of arguments. It determines the number of arguments assuming the C compiler's standard protocol, but the compiler occasionally does things slightly differently for optimization purposes. Such cases are rare. Also, if the stack gets messed up, *tt( )* can become confused. If you suspect a stack crash, you can check for a reasonable stack pointer with *ti( )* before doing a *tt( )*.

10.5 Breakpoints and Single-Stepping

The most versatile way to monitor and debug program execution is to use breakpoints and single-stepping.

A breakpoint can be set at any instruction. When that instruction is executed by an eligible task (as specified with the *b*( ) command), Vx960 suspends the task that was executing and displays a message on the console. At this point, you can examine the task's registers, do a task trace, etc. The task can then be deleted, continued, or single-stepped.

If you single-step a task, the task executes one instruction, then Vx960 suspends the task again, and displays all the task's registers and the *next* instruction to be executed by the task.

The following commands are available for this purpose:

b *adr* [, *task* [, *n*]] Set a breakpoint for *task* at *adr*. If *task* is omitted, the breakpoint is set for all tasks. If *n* is specified, the break will not occur until the *n*th time the breakpoint is hit by an eligible task. *b*( ) with no arguments displays a list of currently active breakpoints.

c [*task*] Continue from a breakpoint or single-step. The task continues execution, starting with the instruction where the breakpoint was located. If *task* is not specified, the current task is continued. If the task is not at a breakpoint, nothing happens.

cret [*task*] Continue until return. This routine places a breakpoint at the return address of the current subroutine of the given task, then continues execution of that task. When the breakpoint is hit, information about the task will be displayed in the same format as in single-stepping. The breakpoint will be removed automatically when the breakpoint is hit, or if the task first hits some other breakpoint. If the task is missing or 0, the current task is assumed. This is useful for examining return values.

bd *adr* [*task*] Delete a specific breakpoint. If *task* is missing, the breakpoint will be removed for all tasks. If the breakpoint applies to all tasks, removing it for just a single task will have no effect. It

must be removed for all tasks, then set for just those tasks desired.

bdall [*task*] — Delete all breakpoints for a task. If *task* is missing, all breakpoints for all tasks are removed.

db [*task*] — Sets a data breakpoint. If the architecture allows it, this routine adds the breakpoint to the list of breakpoints and sets the hardware data breakpoint register(s).

s [*task*] — Single-step a task that is suspended (at a breakpoint for instance). The specified (or current) task will execute one instruction, then display the task's registers and the next instruction to be executed.

so [*task*] — Single-step, but step over a subroutine. This command single-steps a task that is suspended, but, if the next instruction is a call, calls, callx, bal, or balx, it breaks at the instruction following the subroutine call instead. This routine is useful to step over a subroutine, rather than single-step all the way through it.

10.5.1 Unbreakable Tasks

A task can be set *unbreakable*, in which case breakpoints that otherwise would apply to that task are ignored. Tasks are set breakable or unbreakable when they are spawned, or with the routines *taskOptionsSet*( ) and *taskOptionsGet*( ). Several Vx960 tasks set themselves unbreakable, including the shell, the exception support task *excTask*( ) and several network related tasks.

10.5.2 Spawning to Break

There is an important distinction between routines that are executed directly from the shell (by just typing their name) and routines that are spawned as separate tasks (using *sp*( ) or *taskSpawn*( )). When executed directly from the shell, the routine operates in the shell's task context, using the shell's stack, etc. When a routine is spawned, it gets its own task context, its own stack, and its own TCB.

It is not possible to place breakpoints or to single-step in the shell, because, if you could, all control of the system would be lost once the shell was suspended – the shell is the tool that you use to execute commands. This is important because when you execute a routine directly from the shell, that routine executes in the shell's context. Vx960 *always* ignores breakpoints in the shell, or in routines invoked directly from the shell.

To use breakpoints on a routine, that routine *must* be spawned with its own context. The easiest way to do this is with the *sp( )* command, which spawns it with default parameters.

For example, if you have a routine called *hello* which consists solely of a call to *printf( )* to display "hi there", you might enter the following sequence:

```
-> b printf                                                    ❶

-> hello                                                       ❷
hi there

-> sp hello                                                    ❸
Break at 0x1234: _printf      Task: 0x3abcd (t25)

-> tt                                                          ❹
1234   hello + 1c: printf (1500)
```

In the above sequence:

❶ sets a breakpoint at the beginning of the routine *printf( )*.

❷ executes *hello( )* directly from the shell and the breakpoint is not encountered, since the routine is executing in the *shell's* task context and the shell cannot be "broken."

❸ spawns *hello* as a separate task with its own context, and it does therefore hit the breakpoint.

❹ shows that the task is at the beginning of *printf( )*, called from the main routine of *hello( )*.

Notice that *sp( )* spawns the task with an arbitrary task ID (in this case 0x3abcd) and assigns a default task name (in this case t25).

10.6 Disassembler

Vx960 contains a symbolic disassembler. The command to disassemble is:

`l` [*adr*[, *n*]] — List *n* disassembled instructions, starting at *adr*. If *n* is 0 or not given, the *n* from a previous *l*( ) is used. If *adr* is 0, *l*( ) starts from where the previous *l*( ) stopped, or from where an exception occurred, if there has been an exception fault or a breakpoint since the last *l*( ) command.

The disassembler will use any symbols that are in the system symbol table. If an instruction whose address corresponds to a symbol is disassembled (the beginning of a routine, for instance), the symbol is shown as a label in the address field. Symbols are also used in the operand field. The following is an example of disassembled code.

```
                    _l:
1230    5c281611        mov     g1, r5
1234    0b0004a4        bal     dbgInstrPtrAlign.lf
1238    3a04200e        cmpibe  0, g0, 0x001244
123c    92803000        st      g0, _dbgNextToDsm
1244    3a01600e        cmpibe  0, r5, 0x001250
1248    92283000        st      r5, _dbgDfltCnt
1250    90803000        ld      _dbgNextToDsm,  g0
1258    90883000        ld      _dbgDfltCnt,  g1
1260    09000030        call    _dbgList
1264    92803000        st      g0,_dbgNextToDsm
```

This example shows the beginning of the routine that implements the *l* shell command itself. The routine moves the second argument to *l* (passed in register *g1*), into a local register, does a "branch-and-link" leaf procedure subroutine call to align the debug pointer, it conditionally stores the optional arguments, calls the routine that actually lists the code, and finally updates the next address to disassemble for subsequent invocations. Notice that any symbols defined in C (routine and variable names) have an "_" prepended by the compiler. If the routine has an alternate "leaf procedure" entry point added as an optimization by the compiler, the symbol name will have a ".lf" appended by the compiler. (The loader resolves calls to such routines into "bal" instructions.)

10.7 Exception Faults

Vx960 traps all exceptions which can be generated by the i960 CPU family. These exceptions include bus errors, illegal instruction types, divide by zero, integer overflow, etc. When one of these is encountered, Vx960 displays a message on the console terminal and simply suspends the task that was executing when the exception occurred. The rest of the system, including other application tasks and the shell, goes on running as usual.

To find out where the task was when it bombed, do a task trace. Often, you'll find enough information there to pinpoint the problem quickly (a routine called with insufficient arguments, or illegal arguments, for instance).

Once a task has died in this fashion, it is usually not practical to continue the task. When you are through examining the task's data, registers, stack, etc., you can either delete the task (using *td*( )) or restart it (using *taskRestart*( )). If you delete it, it can be restarted later by spawning.

If the shell, or any routine which you call directly from the shell, generates an exception, it is treated specially. This is because you will lose control of the system if the shell is suspended. If this happens, a task trace of the shell is displayed automatically and the shell is restarted. At this point, the best course of action may be to spawn the offending routine as a separate task, and debug it in a context of its own.

10.8 Miscellaneous Useful Utilities

In the course of debugging, there are many utilities which can be useful, even though they may not be functions that are normally considered part of a debugger.

10.8.1 Task Control

You can find the current state of a task with the *i*( ) or *ti*( ) commands. In order to change that state, you can use one of the following:

ts *task* Suspend a task.

tr *task* Resume a suspended task.

td *task* Delete a task.

The normal way of starting tasks is with the following commands. See taskLib(1) for details.

taskSpawn *name, priority, options, stackSize, entryAdrs* [*, arg1, ...arg10*]

Spawn a C routine as a task. Normally, *entryAdrs* will be symbolic. For example:

```
id = taskSpawn "tMyFunc", 50, 0, 10000, myFunc, 123
```

This would spawn the C routine *myFunc*( ) as a task, and pass it the parameter 123.

sp *entryAdrs* [*, arg1, ...arg9*]

Spawn a C routine as a task, with default parameters (*name* = tn, *priority* = 100, *stackSize* = 20000). For example:

```
-> sp myFunc, 123
```

10.8.2 Task Error Status Values

Each task has an *error status* value, which is an integer value that reflects any system errors that may have occurred during task execution. To see a task's status, type *i*( ) (for "info") from the shell and look in the ERRNO column for the hexadecimal status value.

The following command is useful for translating status values to more human-readable error messages:

printErrno *errno* Display the message for the status value *errno*.

See the manual entry errnoLib(1) for more details.

10.8.3 Memory Information

The following command gives information about the current state of memory usage:

memShow [1] This routine prints to standard output the total amount of free and allocated space in the memory pool, the number of free and allocated fragments, the average free and allocated fragment sizes, and the maximum free fragment size. Current as well as cumulative values are given. An argument of 1 also displays the free list of the system partition.

10.8.4 Display and Modify Memory

Two commands are available to examine and modify the contents of memory:

d [*adr*[, *nwords*]] Display *nwords* of memory in hexadecimal and in ASCII format. The *d*( ) command displays a block starting from *adr*, which can be a number, variable, routine, etc. For example:

```
-> d &myStructure
```

would display memory starting with *myStructure* (which must be in the system symbol table). The command *d*( ), with no arguments, displays the next memory block, starting from where the last *d*( ) completed.

m *adr* Modify memory starting at *adr*. The *m*( ) command displays successive words in memory on the terminal; you can change each word by typing a new hex value, leave the word unchanged and continue by typing RETURN, or return to the shell by typing a dot (.).

10.8.5 Display and Modify Registers

Several commands are available to examine and modify register contents. If no task ID is passed to these subroutines, then the current task ID is used.

rn [taskID] return the contents of local register number *n*. The commands **r3** through **r15** are available.

gn [taskID]	return the contents of global register number *n*. The commands g0 through g14 are available.
acw [taskID]	return the contents of the task's arithmetic control word register.
pcw [taskID]	return the contents of the task's process control word register.
pfp [taskID]	return the contents of the task's previous frame pointer register.
rip [taskID]	return the contents of the task's previous return instruction pointer register.
fp [task ID]	return the contents of the task's frame pointer register.
tcw [taskID]	return the contents of the task's trace control word register.
tsp [taskID]	return the contents of the task's task stack pointer register.
mRegs [taskID]	modify registers. The *mRegs* command sequentially prompts you for new values for a task's registers starting with *pfp*. All numbers are entered and displayed in hex.

10.8.6 Symbol Look-up

lkup *string*

Display a list of all symbols in the system symbol table whose names contain *string*. For example, the command:

```
-> lkup "dsm"
```

would display a list containing routines and declared variables containing the string "dsm" as in the following:

```
_dsmData        0x00049d08 text
_dsmNbytes      0x00049d76 text
_dsmInst        0x00049d28 text
mydsm           0x003c6510 bss
```

Case is significant, but position is not (*mydsm* would be shown but *myDsm* would not).

10.8.7 Repeat and Period

Sometimes it is useful to set up a routine to run repetitively. Two commands are available to do this:

repeat *n, func, [arg1, ... arg8]*

Spawn a task that calls *func( ) n* times, with the specified arguments. There will be no delay between calls.

period *secs, func, [arg1, ... arg8]*

Spawn a task that calls *func( )* every *secs* seconds with the specified arguments.

These commands spawn tasks whose entry points are *_repeatRun* and *_periodRun*, which you can see with the *i( )* command. The tasks may be controlled or stopped as described previously under **10.8.1 Task Control**.

10.9 Tips

Debugging with an environment as rich as Vx960 is difficult to describe in a cut-and-dried fashion. Inevitably, each user will come up with an individual way of doing things. The following are some useful tips, for starters.

10.9.1 Task Information: *i( )*

The *i( )* command is perhaps the most commonly used command to get a quick idea of what's going on. By spawning it as a task that runs periodically (using *period( )*), you can set up the shell's terminal to show status every few seconds. If nothing seems to be happening, *i( )* is usually the best place to start.

10.9.2 Where to Set Breakpoints

Although breakpoints can be set anywhere, the most common place to start is generally at the beginning of a subroutine. For example, if you want to check quickly whether a particular routine, *square*( ), is working properly, you might execute the following sequence of commands:

```
-> b square                                                    ❶
value = 0
break at 0x1234: _square    Task: 0x3abcd (t252)

-> tt                                                          ❷
1010: spnTsta + 0x10    myTask ()
2468: myTask + 0x3f0    square (0c)
value = 0

-> cret                                                        ❸
pfp:    3acea0  sp :    3acf70   rip:      3c0fe0   r3 : eeeeeeee
r4 :  eeeeeeee  r5 :  eeeeeeee   r6 :    eeeeeeee   r7 : eeeeeeee
r8 :  eeeeeeee  r9 :  eeeeeeee   r10:    eeeeeeee   r11:   3accf0
r12:    3c0fc0  r13:  eeeeeeee   r14:    eeeeeeee   r15: eeeeeeee
g0 :        90  g1 :    1f0000   g2 :      623bc    g3 :        1
g4 :         a  g5 :         a   g6 :     3faae0   g7 :   3c0fbb
g8 :         0  g9 :         0   g10:           0   g11:        0
g12:         0  g13:    3fa985   g14:           0   fp :   3acf20
pcw:  d86088ff  acw:      9102   tcw:    f001ff81

-> 0x90                                                        ❹
value = 144 = 0x90

-> c                                                           ❺
value = 0
break at 0x1234: _square    Task: 0x3abcd (t252)

-> bdall                                                       ❻
value = 0

-> c                                                           ❼
value = 0
```

In the above sequence:

❶ sets a breakpoint at the beginning of routine *square*( ). If there is some task running that will call *square*( ), the breakpoint will be hit.

❷ is a task trace showing how *square*( ) was called, with a parameter of 12 (0x0c).

❸ is a return of *square( )* which hits a breakpoint, and displays the registers exactly as if it had been a single-stepped task. Since the C calling convention on the 80960CA series is to return a function value in g0, you can see that *square( )* returned a value of 0x90.

❹ uses the shell to convert 0x90 to the decimal value we really prefer, and since 12**2 = 144, *square( )* seems to work properly.

❺ continues the task, calling *square( )* and causing a breakpoint.

❻ deletes the breakpoint since presumably you're now satisfied that *square( )* is working properly.

❼ continues again.

10.9.3 Stack Crashes

Vx960 provides no protection against stack crashes. Such crashes occur when a task uses more stack than has been allocated for it, thus running into and writing over some other task or data. It generally manifests itself as a crash of some task other than the offending one, or other inexplicable behavior.

Before Vx960 starts a task (via *sp( )* or *taskSpawn( )*), it writes 0xeeee over the entire stack allocated to that task. You can tell whether a task has ever overrun its stack (or how much stack the task has ever used) by calling *checkStack( )*. If the stack has overflowed or come close to it, spawn the task again with a bigger stack size.

10.9.4 System Crashes, or "Power Corrupts"

The Vx960 system does little to restrain tasks from global damage to the system or other tasks. This offers the advantage of speed, simplicity, and ability to run on low-level hardware. However, any task can crash the entire system, quickly and totally. A few of the possible ways to do this are:

- Overwriting the code, data, or stack of some other task (or even of the system or shell).
- Overwriting interrupt vectors.

- Taking an important semaphore and never returning it.
- Operating at a high priority (or with interrupts locked out) and never relinquishing the CPU.

If the system is crashing in such a way that you are losing control of it (no response from the shell, I/O messed up, etc.), the best plan of attack is to reboot the system and work "forwards," setting some breakpoints and trying to localize the problem that way.

Alternatively, even after Vx960 dies, you may still be able to get to the ROM monitor (depending on your hardware system) by typing ^X. Debugging from there is primitive, but possible. If this doesn't work, it means either that interrupts are locked out, the console's interrupt vector has been overwritten, or the serial driver's code has been overwritten.

Appendix A

Directories and Files

The following is a summary and description of the various Vx960 directories and files. Figure A-1 shows a schematic diagram of the Vx960 directory structure.

bin Directory containing the executable binaries of the Vx960 tools that run on the host. This directory contains host-architecture-specific subdirectories, such as **sun3** and **sun4**. You may want to add the appropriate subdirectory to your shell-environment search path to facilitate the use of these tools.

config

Directory containing all the files used to configure and build particular Vx960 systems. It includes source to all Vx960 I/O drivers, system-dependent modules, and some user-alterable modules. These files are organized into several subdirectories: the subdirectory **all**, which contains modules common to all implementations of Vx960 (system-*independent* modules), and a subdirectory for each port of Vx960 to specific target hardware (system-*dependent* modules).

config/all

Subdirectory containing all system-independent Vx960 I/O drivers and system configuration modules. The object modules from the sources in this directory are collected in the **gar960** format library **/usr/vx/lib/***arch***/config.a**, where *arch* is the architecture type for your target CPU. The directory includes the following files:

Makefile

Makefile for compiling Vx960 system-independent I/O drivers and system configuration modules.

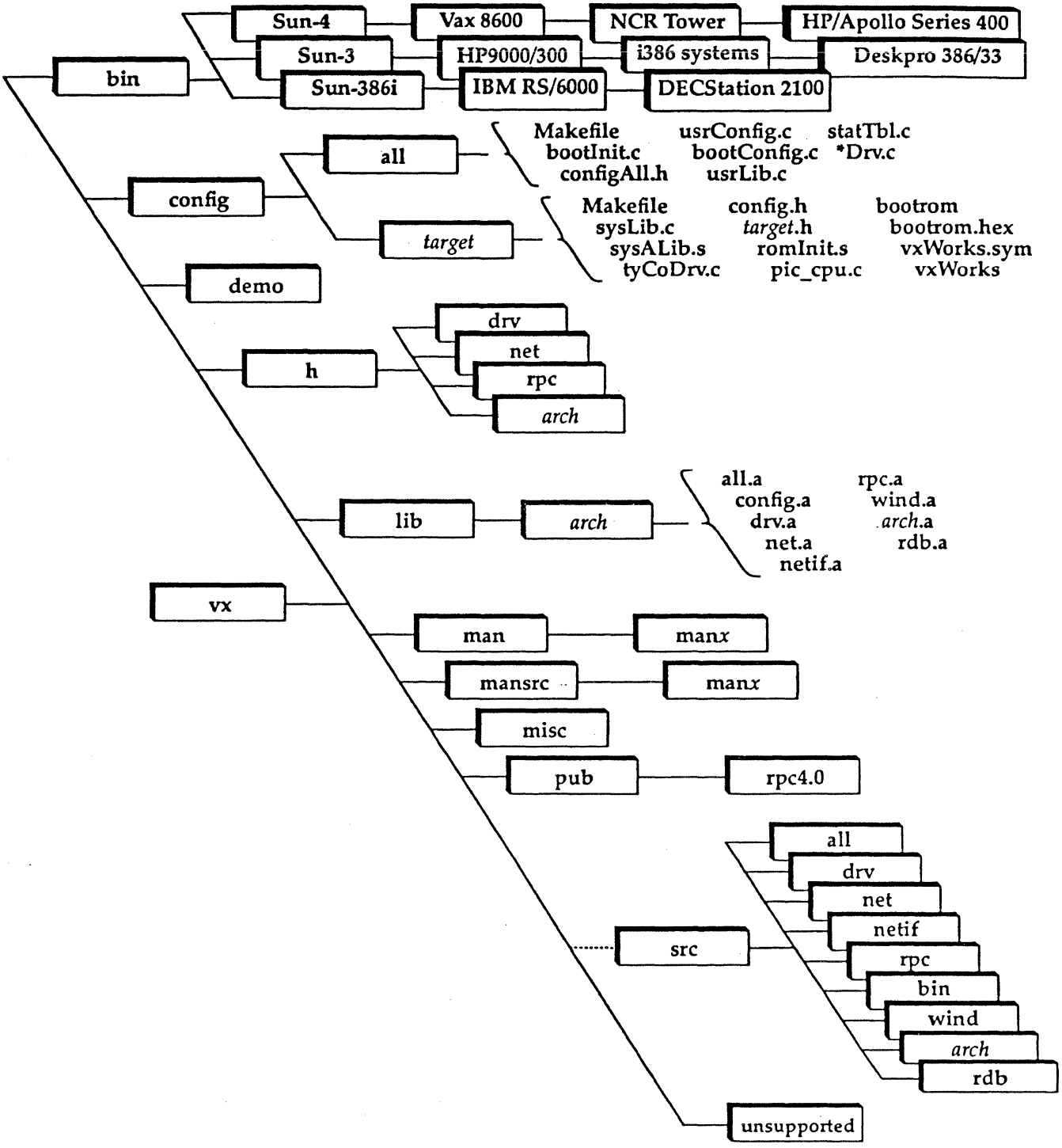


Figure A-1. Vx960 Directory Tree

configAll.h
Generic header file used in the target directories.

usrConfig.c and **bootConfig.c**
Source of the configuration module for a Vx960 development system (**usrConfig.c**), and a configuration module for the Vx960 boot ROM (**bootConfig.c**). Configuration files can be tailored as described in **8. Configuration.**

usrLib.c
Library of routines designed for interactive invocation, which can be modified or extended if desired.

bootInit.c
System-independent boot ROM facilities.

statTbl.c
Source of the error status table. It contains a symbol table of the names and values of all error status codes in Vx960. This table is used by the routine *printError( )* for translating error status codes into meaningful messages.

memDrv.c
Driver for accessing memory as a pseudo I/O device.

ramDrv.c
Driver for emulating a disk in memory.

config/*target*
The other subdirectories of the **config** directory contain the system-dependent modules for each port of Vx960 to a particular target CPU. Each of these directories includes the following files:

Makefile
Makefile for creating boot ROMs and the Vx960 system image for that target.

sysLib.c and **sysALib.s**
Two source modules of system-dependent routines.

tyCoDrv.c
A driver for on-board serial ports.

config.h
Header file of hardware configuration parameters.

target.**h**
Header file for the target board.

romInit.s
Assembly language source for initialization code that is the entry point for the Vx960 boot ROMs and ROM-based versions of Vx960.

pic_cpu.c
"Picture" of the target used by the jump program to display jumper setup.

bootrom
Object module containing the Vx960 boot ROM code.

bootrom.hex
ASCII file containing the Vx960 boot ROM code in Intel Hex format, suitable for downloading over a serial connection to a PROM programmer.

vxWorks and vxWorks.sym
Complete, linked Vx960 system binary (**vxWorks**), and its symbol table (**vxWorks.sym**) created with the supplied configuration files.

demo
This directory contains sample application modules for demonstration purposes, including both the source and the compiled object modules ready to be loaded into vxWorks.

demo/1
Directory containing a simple introductory demo program as well as a server/client socket demonstration.

demo/dg
Directory containing a simple datagram facility, useful for demonstrating and testing datagrams on Vx960 and/or other TCP/IP systems.

demo/xsprites
Directory containing a SUN RPC demonstration. These programs use the X Window System.

h Directory containing all the header (include) files supplied by Vx960. Your application modules need to include several of them to access Vx960 facilities. The **h** directory also contains the following subdirectories:

h/drv Directory containing hardware-specific headers for drivers, etc.

h/net Directory containing all the internal header (include) files used by the Vx960 network. Network drivers need to include several of these headers, but no application modules should need them.

h/rpc Directory containing header files which must be included by applications using the Remote Procedure Call library.

h/*arch* Directory containing architecture-dependent header files.

lib Directory containing all the machine-independent object libraries and modules provided by Vx960. The directory contains architecture-specific subdirectories and the subdirectory **lint**:

lib/*arch*

all.a A gar960 format library containing the object modules that make up the basic Vx960 operating system.

config.a
A gar960 format library containing the object modules whose sources are in the **config/all** directory. These are the Vx960 drivers and several user-configurable modules.

drv.a A gar960 format library containing the object modules for supported controller board drivers.

net.a A gar960 format library containing the object modules for the Vx960 network subsystem.

netif.a
A gar960 format library containing the object modules for the Vx960 network interface drivers.

rpc.a A gar960 format library containing the object modules for the Remote Procedure Call (RPC) library.

wind.a
A gar960 format library containing the object modules for the Vx960 kernel.

arch.a A gar960 format library containing all Vx960 object modules that are architecture-specific.

mansrc
Directory containing the nroff/troff sources for all entries from the Vx960 *Reference Manual*. This directory is divided into subdirectories for libraries, drivers, routines, tools, and target-specific documentation.

man Directory containing nroff-formatted text from the sources in the mansrc directory. Subdirectories mirror those of the mansrc directory. These directories are initially empty, but entries are added as needed whenever vxman is invoked.

pub/rpc4.0
Directory containing public domain SUN source code for the RPC version 4.0.

misc Directory containing miscellaneous support files including makefile templates and documentation macros.

src Directory containing all source files for Vx960. The following additional directories are included.

src/all Directory containing the source files for the basic Vx960 operating system modules. The object modules of these modules are collected in the archive library lib/*arch*/all.a.

src/drv
Directory containing the source code for supported controller board drivers.

src/net
Directory containing the source files for the Vx960 network subsystem modules. The object files of these modules are collected in lib/*arch*/net.a.

src/netif
Directory containing the source files for the Vx960 network interface drivers. The object files of these modules are collected in lib/*arch*/net.a.

src/rpc
Directory containing the source code for RPC which has been modified to run under Vx960. The object files of these modules are collected in **lib/*arch*/rpc.a**.

src/rdb
Directory containing the remote debugger source for **lib/*arch*/rdb.a**.

src/bin
Directory containing the source files for the tools in the **bin** directory. All of the source in **src/bin** is included in every product to allow different UNIX host to be supported by the customer.

src/wind
Directory containing the source code for the Vx960 kernel. The object files of these modules are in **lib/*arch*/wind.a**.

src/*arch*
Directory containing Vx960 source code for architecture-specific modules. The object files of these modules are in **lib/*arch*/*arch*.a**.

unsupported
This directory contains source code that Intel makes available but does not support. This includes user-submitted code, as well as obsolete board support packages and other discontinued material.

Appendix B

Memory Layout

Figure B-1 shows a typical layout of a Vx960 system in memory on a 80960 target CPU. The actual layout depends on the specific board, processor architecture, and Vx960 configuration. For the details of the Vx960 memory layout, you should consult the files, **config.h** and **sysALib.s**, for your target.

80960CA processors have 1K (1024) bytes of on-chip SRAM that you can configure to use as appropriate for your application. When the processor is configured for more than five cached register sets, the additional sets are stored in some of this on-chip memory. The on-chip memory may also optionally contain interrupt vectors, the NMI vector, and DMA registers. The default Vx960 configuration doesn't use any of this space so it is available for application use.

The Initialization Boot Record (IBR, 80960CA) or the Initial Memory Image (IMI, 80960KA/KB, and 80960SA/SB) are also defined by the architecture to reside at fixed addresses within the Vx960 memory space. These areas are used for processor initialization at a cold start or system reset. These are defined in **romInit.s** for boot ROMs and ROM resident versions of Vx960.

Refer to your *80960CA User's Manual, 80960KB Programmer's Reference Manual,* and the *80960SA/SB Reference Manual* for more complete information on the processor defined memory map.

The other areas of the memory used by Vx960 are defined relative to the local (on-board) memory base address, LOCAL_MEM_LOCAL_ADRS, as defined in the file **config.h**. If this base is 0, the first 1K addresses on the 80960CA references the on-chip SRAM. If this base starts in some other region, it does not overlap the on-chip address ranges. These other areas in Figure B-1 are labeled as follows:

BP Anchor Anchor for the backplane network if there is shared memory on the board (does not apply to embedded designs).

Boot Line ASCII string of boot parameters.

Exception Message

ASCII string of the fatal exception message.

System Image Entry point **_sysInit** is board dependent, but is typically LOCAL_MEM_LOCAL_ADRS+0x1000.

The System Image also includes these architecture dependent data structures (these are shown as part of the text segment but may actually be part of the data segment on some systems; developers may reorganize **sysALib.s** to suit their needs) the sizes are defined below, but their locations depend on System Image size and alignment.

- **prcb** - i960 processor control block contains pointers to other architecture defined data structures and initialization data, 40 bytes on 80960CA, 176 bytes on 80960KA/KS and 80960SA/SB.
- **_ctrlTable** - 80960CA control table contains values for on-chip control registers, 112 bytes.
- **_systemAddressTable** - 80960KA/KB or 80960SA/SB-system address table (SAT only) points to the system procedure table and the fault table with much unused (preserved) space, 176 bytes.
- **intTable** - i960 interrupt vector table; 248 vectors x 4 bytes each plus 8-byte pending, interrupt section, plus a 1-byte pending priorities section.
- **_faultTable** - i960 fault table, 16 entries x 8 bytes each.
- **_sysProcTable** - system-procedure-call table, and/or fault-procedure table, 12 bytes reserved, a 4-byte pointer to the supervisor stack, 24 preserved bytes and up to 260 entries x 4 bytes each.
- **_intStack** - i960 interrupt stack, size defined in **config.h** and/or **sysALib.s**.

- **_supervisorStack** - initial stack used by *usrInit*() until *usrRoot*() is allocated a stack; also used for the fault and trace stack; can be used as the system call stack, but no processor system calls are defined in Vx960; size defined in **config.h** and/or **sysALib.s.**

System Memory Pool

The pool of free memory from which dynamic memory resources can be allocated, including stacks, semaphores, communication buffers, etc. The end of the free memory pool for a given board is returned by *sysMemTop*().

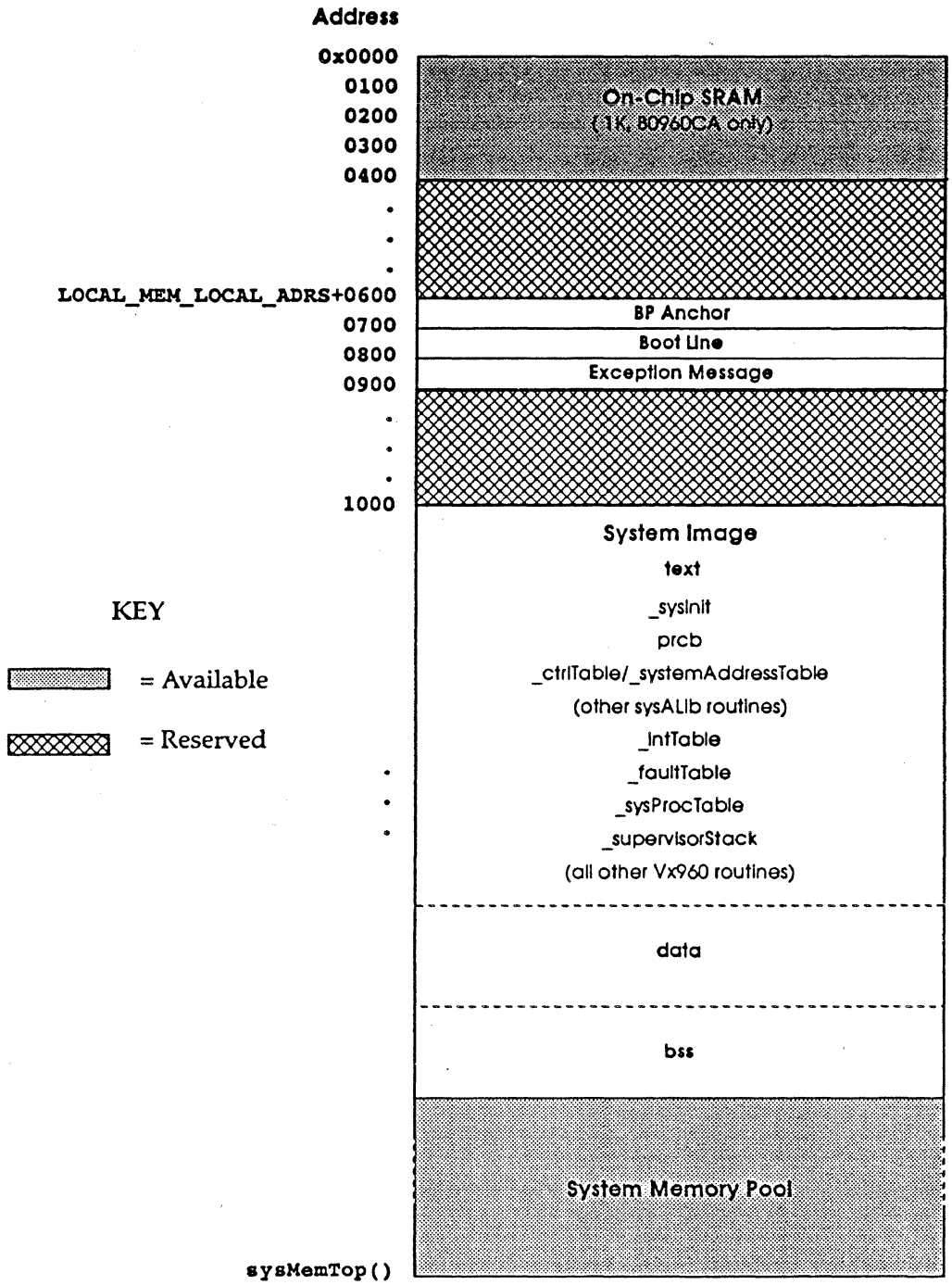


Figure B-1. Vx960 System Memory Layout (Intel 80960 Microprocessors)

Appendix C

Coding Conventions

C.1 Introduction

The conventions specified define the Intel standard for all C code and accompanying documentation included in the Vx960 source. The conventions are intended to encourage higher quality code: every source module is required to have certain essential documentation, and the code and documentation is required to be in a format that has been found to be readable and accessible.

The conventions are also intended to provide a level of uniformity in the code produced by different programmers. Besides making the code look more professional, uniformity allows programmers to work on code written by others with less overhead in adjusting to stylistic differences. Also it allows automated processing of the source: tools can be written to generate manual entries, module summaries, change reports, etc.

These conventions are divided into the following categories:

- Module Layout
- Subroutine Layout
- Code Layout
- Naming Conventions
- Style

C.2 Module Layout

A module is any unit of code that resides in a single source file. Thus a module may be a UNIX tool, a library of routines, or an applications task. The conventions in this section define the standard for the header which must come at the beginning of every source module.

The module header consists of the blocks described below. The blocks are separated by one or more blank lines and should contain no blank lines within the block. This facilitates automated processing of the header.

- **Title:** The title consists of one line containing the module name followed by a short description. The module name must be the same as the module file name. This line becomes the title of automatically generated manual entries and indexes.
- **Modification History:** Each entry consists of the version number, date of modification, initials of the programmer who made the change, and a complete description of the change. The version number is a two-digit number and a letter (e.g., "03c"). The letter is incremented on small changes, and the number is incremented on major changes, especially those that materially affect the module's external interface.
- **General Module Documentation:** This consists of a complete description of overall module purpose and functioning, and especially the external interface.
- **Includes:** This consists of the includes required by the module.

The exact format of these blocks is elaborated in the following example of a standard module header:

```
/* fooLib.c - foo subroutine library */

/*
modification history
--------------------
01b,15feb86,dnw  added routines fooGet and fooPut;
                 added check for invalid index in
                   fooFind.
01a,10feb86,dnw  written.
*/
```

```
/*
DESCRIPTION
This module is an example of the Intel
C coding conventions.
...
*/

#include <vxWorks.h>
#include "fooLib.h"
```

C.3 Subroutine Layout

The following conventions define the standard layout for every subroutine in a module.

Each subroutine is preceded by a header of documentation that consists of the following blocks. There should be no blank lines in the header but each block should be separated with a line containing a single asterisk (*) in the first column.

- Banner: This is a slash character followed by asterisks across the page.
- Title: One line containing the routine name followed by a short description. The routine name in the title must match the declared routine name. This line becomes the title of automatically generated manual entries and indexes.
- Description: Complete description of what the routine does and how to use it.
- Returns: The word "RETURNS:" followed by a description of the possible result values of the subroutine.

Immediately following the subroutine header should be the subroutine declaration itself. Each argument to the routine must be described with a comment. This declaration becomes the synopsis in automatically generated manual entries.

The exact format of the subroutine header and declaration is shown in the following example of a standard subroutine layout:

```
/**********************************************
*
* fooGet - get an element from a foo
*
* This routine finds the element of the specified
* index in the specified foo. The value of the element
* found is copied to <pValue>.

*
* RETURNS:
* OK if element found, otherwise ERROR.
*/

STATUS fooGet (foo, index, pValue)
    FOO foo;     /* foo in which to find element */
    int index;   /* element to be found in foo */
    int *pValue; /* where to put value */
    {
    ...
    }
```

C.4 Code Layout

The following conventions define the standard for the graphic layout of C code.

C.4.1 Vertical Spacing

(1) Use blank lines to make code more readable and to group logically related sections of code together. Put a blank line before and after comment lines.

(2) Do not put more than one statement on a line. The only exception is the *for* statement where the initial, conditional, and loop statements may be written on a single line:

```
for (i = 0; i < count; i++)
```

The *if* statement is not an exception: the executed statement always goes on a separate line from the conditional expression:

```
if (i > count)
    i = count;
```

(3) Section braces, { and }, and *case* labels always get their own line.

C.4.2 Horizontal Spacing

(1) Put spaces around binary operators, after commas, and before open parenthesis. Do not put spaces around structure member and pointer operators. Put spaces before open brackets of array subscripts, although if a subscript is only one or two characters long, the space may be omitted.

For example:

```
status = fooGet (foo, i + 3, &value);
foo.index
pFoo->index
fooArray [(max + min) / 2]
string[0]
```

(2) Continuation lines should line up with the part of the preceding line they continue:

```
a = (b + c) *
    (d + e);

status = fooList (foo, a, b, c,
                  d, e);

if ((a == b) &&
    (c == d))
    ...
```

C.4.3 Indentation

(1) Tab stops are every four characters (i.e., 1, 5, 9, ...).

(2) The module and subroutine headers and the subroutine declarations start in column 1.

(3) Indent one tab stop after:

- subroutine declarations
- conditionals (see below)
- looping constructs
- switch statements
- case labels
- structure definitions in a *typedef*.

(4) The *else* of a conditional has the same indentation as the *if*. Thus the form of the conditional is:

```
if ( condition )
    {
    statements
    }
else
    {
    statements
    }
```

(5) The general form of the switch statement is:

```
switch ( input )
    {
    case 'a':
        ...
        break;

    case 'b':
        ...
        break;

    default:
        ...
        break;
    }
```

If the actions are very short and nearly identical in all cases, an alternate form of the switch statement is acceptable:

```
switch ( input )
    {
    case 'a': x = aVar; break;
    case 'b': x = bVar; break;
    case 'c': x = cVar; break;
    }
```

(6) Comments have the same indentation as the section of code to which they refer.

(7) Section braces, { and }, have the same indentation as the code they enclose.

C.5 Naming Conventions

The following conventions define the standards for naming modules, routines, variables, constants, macros, types, and structure and union members. The purpose of these conventions is uniformity and readability of code.

(1) When making up names, keep the following in mind: the code is written once but read many times. Make names meaningful and readable. Avoid obscure abbreviations.

(2) Names of routines, variables, and structure and union members are written with upper and lower case and no underlines. Each "word" except the first is capitalized:

```
aVariableName
```

(3) Names of defined types (via *typedefs*), and constants and macros (via *#defines*), are all upper case with underlines separating the words in the name:

```
A_CONSTANT_VALUE
```

(4) Every module has a short prefix (2-5 characters). The prefix is attached to the module name and all externally available routines, variables, constants, macros, and *typedefs*. (Names not available externally do not following this convention.)

```
fooLib.c            - module name
fooFind             - subroutine name
fooCount            - variable name
FOO_MAX_COUNT       - constant
FOO_NODE            - type
```

(5) Pointer variable names have the prefix "p" for each level of indirection:

```
FOO_NODE  *pFooNode;
FOO_NODE **ppFooNode;
```

C.6 Style

The following conventions define additional standards of programming style:

- **Comments:** Insufficiently commented code is unacceptable code. Serious or repeated violations are grounds for public derision of the perpetrator.
- **Lint:** The **lint** tool is run with every installation to check for possible errors that would not be caught by the compiler. All modules must be coded so as to eliminate all avoidable **lint** complaints. This frequently involves explicit type coercions and use of **lint** directives (VARARGS, ARGSUSED, etc.).
- **Mangen:** The **mangen** tool is used to generate manual entries for every module automatically. All modules must generate valid manual entries by **mangen**. The layout requirements for **mangen** options are documented in the previous pages.
- **Numeric Constants:** Use *#define* to define meaningful names for all constants. Never use numeric constants in code or declarations (except for obvious uses of 0, 1, etc.).
- **Structure Definitions:** Always define structures and unions by a *typedef* declaration. Never use a structure or union definition to declare a variable directly.
- **Passing and Returning Structures:** Always pass and return pointers to structures. Never pass or return structures directly.
- **Return Status Values:** Routines that return status values should return either OK or ERROR (defined in vxWorks.h). The specific type of error is identified by setting the task status value with the routine *errnoSet( )*.

- **Use Defined Names:** Use the names defined in **vxWorks.h** wherever possible. In particular, note the following definitions:
    - Use TRUE and FALSE for boolean compares.
    - Use EOS for end-of-string tests.
    - Use NULL for zero pointer tests.
    - Use FAST for *register* variables.
    - Use IMPORT for *extern* variables.
    - Use LOCAL for *static* variables.
    - Use VOID for routines that return no value.
    - Use FUNCPTR for pointer-to-function types.

Appendix D

Bibliography

D.1 Introduction

This bibliography is a representative collection of resources that will help you become familiar with the concepts of internetworking, operating systems in general, and UNIX in particular. It is only a starting place for further research. If you want more in-depth information on the issues and complexities of these topics, refer to the bibliographies that exist in many of the resources cited here.

Note: Some of the information included in this appendix can be found in the *RFC 1175, FYI on Where to Start - A Bibliography of Internetworking Information.*

D.2 Networking Bibliography

We have divided the networking bibliography into primary and secondary sources. A primary source is a book that is directly applicable to TCP/IP programming. A secondary source is a book that relates to more general networking topics.

D.3 Request for Comment (RFC) of Interest to Beginners

RFC refers to a large number of documents kept on-line at various Internet sites that constitute discussions, general information, and specifications for various aspects of the overall Internet networking and TCP/IP protocol world. In contents they range from a poem to Sun's specification for NFS as Remote Procedure Call routines. Doug Comer talks about how to obtain RFC's on pp. 443-446 of his book *Internetworking with TCP/IP, Volume 1.*

For further exploration, see Comer or the RFC's themselves. Under **Primary Sources** below, the set of DDN protocol handbooks presents a number of the most important RFC's for TCP/IP in a 3-volume handbook format. You can either purchase these from SRI or get them one at a time over the Internet via anonymous ftp. See Comer for details on how to do that.

D.4 Books We Recommend

- Doug Comer's books on TCP/IP. Good introduction.
- W. Richard Steven's book on UNIX Networking programming.
- The Berkeley Kernel book. It provides internals information on the 4.3 BSD Tahoe TCP/IP implementation.

D.4.1 Primary Sources

The following primary sources are arranged in the order of their priority.

- Comer, Douglas E., *Internetworking With TCP/IP: Principles, Protocols, and Architecture,* 382 pp., Prentice Hall, Inc., Englewood Cliffs, NJ, 1991. Second Edition.

Note: A second volume on internals is available as implemented in XINU, Comer's academic UNIX clone. The ISBN for the first volume, second edition is: 0-13-468505-9.

This book provides an overview and introduction to TCP/IP. It contains an overview of the Internet; reviews underlying network technologies; examines the internetworking concept and architectural model; covers the basics of the Internet addressing gateway system and protocol gateways used to exchange routing information; and discusses application level services available in the Internet. It also contains several useful appendices including RFCs, and how to get them, a glossary of Internet terms, and the official DARPA Internet protocols.

- Richard Stevens. *UNIX Network Programming*. Prentice Hall. 1990. ISBN 0-13-949876-1.

  This book is excellent on the ins/outs of application network programming including socket programming, TLI, library routines, security, and other topics like rlogin.

- Leffler, McKusick, Karels, Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison-Wesley. 1989. ISBN 0-201-06196-1.

  This book discusses internals for the 4.3 BSD operating system. Included is a comprehensive discussion of the networking protocols including the TCP/IP stack and how sockets work.

- Tanenbaum, Andrew S., *Computer Networks*, Second Edition, Prentice Hall, Englewood Cliffs, NJ, 1988.

  This book is a reference for computer communications and networking on the academic side. In addition to OSI, some aspects of TCP/IP are discussed.

- Nemeth, Snyder, Seebass. *UNIX System Administration Handbook*. Prentice Hall. 1989. ISBN 0-13-933441-6.

  This book discusses networking system administration on a Berkeley/-Sun system.

- John Corbin. *The Art Of Distributed Applications.* Springer-Verlag. Sun Technical Reference Books. 1991.

  This book discusses how Sun's remote procedure call (RPC) mechanism works. RPC is used by NFS.

- Marshall Rose. *The Simple Book, An Introduction to Management of TCP/IP - based Systems.* Prentice Hall. 1991. ISBN 0-13-812611-9.

  This book focuses on how SNMP works.

- Feinler, Elizabeth J., Ole J. Jacobsen, Mary K. Stahl, and Carol A. Ward, *DDN Protocol Handbook,* 2749 pp. [3 volumes], SRI International, DDN Network Information Center, Menlo Park, CA, December 1985.

  This is a three volume collection of documents (basic TCP/IP RFCs) addressing how to attach computers to the Defense Data Network (DDN) using the Department of Defense (DoD) suite of protocols. The first volume contains official military standard protocols, such as the Transmission Control Protocol/Internet Protocol (TCP/IP), and the File Transfer Protocol (FTP). Volume two includes all of the official Defense Advanced Research Projects Agency (DARPA) protocols. The final volume contains supplementary material of interest to protocol implementors. In addition, the handbook presents general information about the protocol standardization process itself, the agencies involved and their roles, and the means for obtaining further information. Available from SRI International, DDN Network Information Center, 333 Ravenswood Ave., Room EJ291, Menlo Park, CA 94025.

- Stallings, William, *Handbook of Computer-Communications Standards Volume 1: The Open System (OSI) Model and OSI-Related Standards,* Macmillan, New York, NY, 1990.

- Stallings, William, *Handbook of Computer-Communications Standards Volume 2: Local Area Network Standards,* Macmillan, New York, NY, 1990.

• Stallings, William, *Handbook of Computer-Communications Standards Volume 3: The TCP/IP Protocol Suite*, Macmillan, New York, NY, 1990.

This series systematically covers the major standards topics, providing the introductory and tutorial material not found in the actual standards documents. The books function as a primary reference for those who need an understanding of the technology, implementation, design, and application issues that relate to the standards.

D.4.2 Secondary Sources

The following secondary sources are listed in alphabetical order.

• Anderson, Costales, Henderson, and The Waite Group, *UNIX Communications*, 542 pp., Howard W. Sams & Company, Indianapolis, IN, 1987.

UNIX Communications provides a good overview and comprehensive introduction on UNIX mail, the USENET News and UUCP with clear examples.

• Arms, Caroline, *Campus Networking Strategies*, 321 pp., Digital Press, Bedford, MA, 1988.

This book contains a survey of ten colleges and universities that have made or implemented grand plans for networking. The case studies cover the planning process, technical issues, and financing and management of an ongoing service organization. Chapters on protocols and standards, wiring, and national networks provide valuable technical background. A glossary defines frequently used networking terms. This book is a project of the EDUCOM Networking and Telecommunications Task Force (NTTF), a group of research universities engaged in joint programs to support the development of computer networking technology.

• Arms, Caroline ed., *Campus Strategies for Libraries and Electronic Information*, Vol. 3, 404 pp., Digital Press, Bedford, MA, 1989.

This book offers a comprehensive look at planning and implementation of libraries and information systems in higher education. This is volume 3 in the EDUCOM Strategies Series on Information Technology. Order source for EDUCOM members is: pubs@educom.edu. Order source for non-members is: 1-800-343-8321. Order number: ey-cl85e.dp.

- Batt, Fred, *Online Searching for End Users: An Information Sourcebook*, 116 pp., Oryx Press, Phoenix, AZ, 1988.

  This is a source book for computer and information science which includes bibliographies and indexes.

- Connors, Martin, *Computers and Computing Information Resources*, 1271 pp., Gale Research Co., Detroit, MI, 1987.

  This is a guide to approximately 6,000 print, electronic, and live sources of information on general and specific computer-related topics in all disciplines.

- Frey, Donnalyn and Rick Adams, *!%@:: A Directory of Electronic Mail Addressing and Networks*, Second Edition, 284 pp., O'Reilly and Associates, Sebastopol, CA 1990.

  This handbook of electronic mail addressing and networks contains an electronic mail tutorial, short descriptions of networks, and helpful indices of domain names and ISO codes. It also has several useful appendices: second-level domains sorted by organization name, second-level domains sorted by domain name, ISO country codes sorted by country, same sorted by code, and UUCP mail handling.

- Garcia-Luna-Aceves, Jose J., Mary K. Stahl, and Carol A. Ward, Internet Protocol Handbook: *The Domain Name System (DNS) Handbook*, 219 pp., SRI International, Network Information Systems Center, Menlo Park, CA, August 1989.

  This handbook explains the Domain Name System (DNS) and the Internet Host Table. This is volume four of the DDN Protocol Handbook (see Feinler, E., et. al., DDN Protocol Handbook). This volume is divided into two sections. The first section covers the concepts and philosophy of the DNS as discussed in various articles and Requests for Comments (RFCs). The second section focuses on the transition from the Internet Host Table to the DNS. Detailed information on DNS protocol standards and implementations are provided as are guidelines for the establishment and operation of domain name servers. The handbook concludes with a glossary of DNS acronyms. Available from SRI International, Network Information Systems Center, 333 Ravenswood Ave., Room EJ291, Menlo Park, CA 94025.

- Karrenberg, Daniel and Anke Goos, *European R&D E-mail Directory*, 210 pp., European UNIX Systems Users' Group, Owles Hall, Owles Lane, Buntingford, Herts, England, December 1988.

  This book contains a reference of all organizations reachable by EARN and EUNet, the two major European electronic mail networks serving the research and development community. It contains an electronic mail tutorial and organization indexes. For more information, send electronic mail to *euug@inset.uucp*, or call +44 763 73039.

- LaQuey, Tracy L., *User's Directory of Computer Networks*, 653 pp.,Digital Press, Bedford, MA, May, 1990.

  This directory contains detailed lists of hosts, site contacts, and administrative domains, and general information on over 40 major networks. Included are tutorials on the Domain Name System, X.500, and Electronic Mail. An Organization List, which includes universities, colleges, research institutions, government agencies and companies, cross references much of the network and host information presented throughout the directory. Most of the lists and articles are provided or written by Network Information Centers and network contacts. For more information, send electronic mail to **netbook@nic.the.net**.

- McConnell, John, *Internetworking Computer Systems: Interconnecting Networks and Systems*, 318 pp., Prentice Hall, Englewood Cliffs, NJ, 1988.

  An advanced reference series on Internetworking computer systems and computer networks. Includes bibliographical references and index.

- Quarterman, John S., *The Matrix: Computer Networks and Conferencing Systems Worldwide*, 746 pp., Digital Press, Bedford, MA, 1990.

  A successor to the article "Notable Computer Networks" published by the CACM, October 1986, this book contains background material introducing important topics for readers unfamiliar with networks and conferencing systems. It provides descriptions of specific systems, organized geographically, to facilitate discussion of regional history. Maps are included. Syntaxes and gateways are provided for sending mail from one system to another. Access information is given for those wishing to join or research a system. Extensive reference sections are at the end of each chapter including a sixty-page index of programs and protocols, networks and gateways, places and people. For more information, send electronic mail to **matrix@longway.tic.com**.

- Rose, Marshall T., *The Open Book: A Practical Perspective on OSI*, 651 pp., Prentice Hall, Englewood Cliffs, NJ, 1989.

  This is a comprehensive book about Open Systems Interconnection (OSI). In particular, this book focuses on the pragmatic aspects of OSI: what OSI is, how OSI is implemented, and how OSI is integrated with existing networks. To provide this pragmatic look at OSI, the book makes consistent comparisons and analogies of the OSI pieces with the TCP/IP suite of networking protocols.

- Stoll, Clifford, *The Cuckoo's Egg: Tracking a Spy through the Maze of Computer Espionage*, Doubleday, New York, NY, 1989.

  Clifford Stoll, an astronomer turned UNIX System Administrator, recounts an exciting, true story of how he tracked a computer intruder through the maze of American military and research networks. This book is easy to understand and can serve as an interesting introduction to the world of networking. Jon Postel says in a book review, this book "... is absolutely essential reading for anyone that uses or operates any computer connected to the Internet or any other computer network."

- Todinao, Grace, *Using UUCP and USENET: A Nutshell Handbook*, 199 pp., O'Reilly and Associates, Newton, MA, 1986.

  This handbook outlines how to communicate with both UNIX and non-UNIX systems using UUCP and cu. By example it shows how to read news and post your own articles to other USENET members.

D.4.3 Important RFCs

The following list contains RFCs that a new user will find most informative.

- RFC index. This can be obtained from the NIC by sending a message to SERVICE@NIC.DDN.MIL with a subject line of "RFC index". In general, any RFC can be obtained by this method via email. Substitute the RFC number for the X's as follows in the subject line: "RFC XXXX". See Comer for details.

- RFC 1175 - *FYI on Where to Start.* A Bibliography of Internetworking Information.

- RFC 1177 - *FYI on Questions and Answers.* Answers to commonly asked new user questions.

- RFC 1180 - *A TCP/IP Tutorial.* This RFC presents a brief tutorial on TCP/IP basics.
- RFC 1000 - *Request for Comments Reference Guide.* This RFC is a reference guide for earlier RFCs. It presents a list of all RFCs numbered less than 1000 and discusses each in turn.

D.5 Operating System Bibliography

The following is a bibliography for UNIX in particular and operating systems in general.

D.5.1 Books on UNIX Internals

- Bach, Maurice J., *The Design of the UNIX Operating System.* Prentice Hall Inc., 1986. ISBN 0-13-201799-7.

  This manual describes internals of UNIX with emphasis on system V, some information on BSD systems. Good introduction to file system and process concepts. Networking coverage is poor.

- Leffler, McKusick, Karels, Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System.* Addison-Wesley. 1989. ISBN 0-201-06196-1.

  Details of 4.3 BSD system (VAX); i.e., UNIX system internals. Good reference book for practitioners. Good discussion of architectural tradeoffs. Fair amount of information on sockets and network internals; especially TCP.

- *BSD UNIX Programmer's Manual - UPM, USENIX*

  This manual provides the system manual pages (man pages) plus whatever extra documentation is bundled in (traditional UNIX documents like Shell Introduction by Bourne). Per system; e.g., Sun version from Sun. System V versions, from A.T.T. As available from Usenix Association/ 4.3BSD version, Nov. 1986.

BSD Description follows:

- URM - User's Reference Manual
  1 - commands, cc(1), ls(1),
  6 - games, rogue(6)
  7 - misc.
- PRM - Programmer's Reference Manual
  2 - kernel, read(2), fork(2)
  3 - libraries, fread(3), strcpy(3)
  4 - devices, rk(4), rflop(4)
  5 - files, termcap(5), dbx(5), passwd(5), a.out(5)
- SMM - System Manager's Manual - system administration how-to
  8 - system administration, fsck(8)
- USD - User Supplementary Documents
  various UNIX documents/guides/tutorials
- PS1 - Programmer Supplementary Docs, Part 1.
  papers focused on programming, e.g.,
  Introduction to 4.3BSD IPC
  Advanced 4.3BSD IPC
  Debugging with DBX
- PS2 - Programmer Supplementary Docs, Part 2.
  some papers on historical aspects of UNIX, critical papers by Thompson, Ritchie, and Kernighan, e.g., UNIX I/O System - device driver basics.
- Master Index
  Note the IPC papers in PS1. These talk about socket level programming issues on a BSD system.

- Nemeth, et. al., *UNIX System Administration Handbook.* Prentice Hall., 1989. ISBN 0-13-933441-6.

  This handbook provides good coverage of UNIX System Administration topics. More BSD oriented than System V. Topics include uucp, mail/sendmail, network administration, news, kernel configuration, and other system administration chores.

- Egan and Teixeria. *Writing a UNIX Device Driver.* Wiley and Sons. 1988. ISBN 0-471-62811-5.

  This manual provides discussion and examples concerning writing device drivers on UNIX systems.

- AT&T and Sun manuals should be consulted. Both have information at this point on how to do things like configure kernels and program device drivers. The AT&T books are for sale at technical bookstores.

D.5.2 Operating Systems In General

- Tanenbaum, Andrew S., *Operating Systems - Design and Implementation.* Prentice Hall. ISBN 0-13-637406-9. 1986.

  This is the "Minix" book. Minix is intended as a version 7 UNIX system call compatible o.s. running on the IBM-pc. Provides basic operating system principles and illustrates them via architectural discussion and source from Tanenbaum's Minix system, v7 UNIX as redesigned from the ground up by Andrew T and friends at the Free University in Amsterdam. Source is included in the book; code and binaries available for IBM-pc's. Good introductory o.s. text and much more. Especially good at the moment due to the fact that Minix is designed as a distributed operating system; i.e., according to the client-server model as a set of processes. Minix, unlike UNIX, is architectured according to modern O.S. design architecture precepts.

- Comer, Doug. *Operating System's Design. The XINU Approach.* 1984. ISBN 0-13-637539-1. Prentice Hall.

- Comer, Doug. *Operating System's Design. Volume II Internetworking with XINU.* 1987. ISBN 0-13-637414-X. Prentice Hall.

  Book I introduces Xinu, another pedagogical UNIX-alike operating system. Good introductory text for operating system study. Book II introduces internet networking basics as implemented in Xinu. Presentation of material in both books is through code examples which makes them valuable. Book II lacks information on TCP but provides a good grasp of the lower networking layers. Comer's presentation of networking concepts is excellent.

- Peterson and Silbershatz. *Operating System Concepts.* ISBN 0-201-06198-8. Addison-Wesley. 2nd. Edition. 1985.

  This manual is a standard, introductory college text for operating systems.

D.5.3 UNIX O.S. and O.S. Related books/standards/magazines

- Harbison, Samuel P., Steele Guy L, *C: A Reference Manual.* Second Edition. Prentice Hall, Inc., Englewood Cliffs, New Jersey. 1987.

  This manual includes a complete description of the full C language, the Draft Proposed ANSI C language, and over 180 standard run-time library functions.

- Kernighan and Ritchie, *The C Programming Language.* Second Edition. Prentice Hall. ISBN 0-13-110163-3. 1988.

  This book is the new edition of a UNIX classic. It covers the C programming language and includes material from the Draft-proposed ANSI C standard. The grammar in back is (and has been) the last word in C syntax. The grammar is now Yacc compatible. This book is an important reference work.

- Kernighan/Pike. *The UNIX Programming Environment.* Prentice Hall. ISBN 0-13-937681-X. 1984.

  This manual tries to explain all at the command level, file system basics, shell usage, filters, grep, awk, sed, shell programming, stdio, system calls, nroff/ms. General introduction to UNIX and still quite valuable. Good reference work.

- Posix Standard 1003.1

  ANSI/IEEE Standard Portable Operating System Interface On Computer Environments. 1988. ISBN 1-55937-003-3.

  IEEE
  345 East 47th Street
  New York, New York 10017

  Important. Operating system interface standard. Basically UNIX but

leaves out networking/graphics devices. Started out as version 7 and then added various things from BSD, System V realms (e.g, job control, vs. fcntl(2)). Library interface is basically left to ANSI C standard. Has essential UNIX bibliography in back.

- System V Interface Description. (SVID)

  A.T.T. 1986. 3 Volumes. ISBN 0-932764-10-X. A.T.T. spec for System V.

  A.T.T Customer Info Center
  Attn. Customer Service Rep.
  P.O. Box 19901
  Indianapolis, Indiana 46219

  1-800-432-6600.

- Usenix Association Conference Proceedings

  UNIX-related papers presented at various Usenix conferences held in the last ten years. Often new developments in UNIX are presented at these conferences. In recent years a fair number of papers have been presented that touch on issues of distributed operating systems (like MACH) and distributed programming.

  Usenix Association
  2560 Ninth Street, Suite 215
  Berkeley, CA 94710
  Manual Order Dept:
  415/528-8649

  $40 for ;login: subscription
  7 volume-set, UNIX reference manual@ $68

- UNIX Review

  Magazine issued monthly. Covers various topics of interest in the UNIX world. Miller Freeman Publications, 500 Howard ST., San Francisco, California 94105. (415)-397-1881.

- Embedded Systems. Miller Freeman Publications, 500 Howard ST., San Francisco, California 94105. (415)-397-1881.

  Magazine devoted to programming and architectural issues for embedded system design.

There are various other magazines and journals that touch on operating system issues, *Sun Expert, Sun World, UNIX World,* and various IEEE and ACM journals.

D.6 GNU/960 Documentation

The following is a list of the GNU/960 documentation currently available. You received copies of this documentation with your Vx960.

- The *GNU/960 Manual* contains the following documentation:
  - *GNU/960 Tools Release Notice*

    This manual provides an overview of the highlights of this release. It covers the differences between this release and previous releases, and it includes information on fixes to known bugs.
  - *Installing the GNU/960 Tools*

    This manual gives detailed step-by-step installation information for the GNU/960 tools (source, binaries, and documentation)
  - *Using the GNU/960 Tools*

    This entry-point document describes the history and usage of the tools, gives a summary overview of each command in the tool set, and explains the concepts and process of developing an application using the GNU/960 tools.
  - *GNU/960 Tool Development*

    This manual describes how to customize the GNU/960 tools for a variety of purposes, including porting to other hosts.
  - *GNU/960 Reference Manual*

    This manual contains the man pages which provide detailed information on each GNU/960 command. Many man pages also provide feature examples.

The **intro(1GNU)** man page gives an overview of all the GNU/960 tools.

- *Using Intel i960 GNU CC*

  This manual provides in-depth information on gcc960, the GNU/960 C compiler, including detailed descriptions of all command-line options. This is the source for complete reference information on gcc960.

- *Using GDB/960*

  This manual provides in-depth information on gdb960, the GNU/960 debugger, including detailed descriptions of all command-line options. This is the source for complete reference information on gdb960. A gdb960 quick reference is also included.

- *Using GAS/960*

  This manual provides in-depth information on gas960, the GNU/960 assembler, including detailed descriptions of all command-line options. This is the source for complete reference information on gas960.

- *Using GLD/960*

  This manual provides in-depth information on gld960, the GNU/960 linker including detailed descriptions of all command-line options. This is the source for complete reference information on gld960.

- Harbison, Samuel P., Steele Guy L, *C: A Reference Manual*. Second Edition. Prentice Hall, Inc., Englewood Cliffs, New Jersey. 1987.

  This manual includes a complete description of the full C language, the Draft Proposed ANSI C language, and over 180 standard run-time library functions.

D.7 Processor Documentation

The following is a list of the Intel processor documentation currently available.

- *80960CA User's Manual*

  This manual provides detailed programming and hardware design information for the 80960CA. It is written for programmers and hardware designers who

understand the basic operating principles of integrated processors and their systems. Order Number: 270710-001.

- *80960KB Programmer's Reference Manual*

  This manual provides detailed programming information for the Intel 80960KB processor. Order Number: 270567-002.

- *80960KB Hardware Designer's Reference Manual*

  This manual is the definitive hardware reference guide for system designs using the 80960KB processor. It is written for hardware designers as a guideline for developing microprocessors systems. These hardware designers should be familiar with the operating principles of microprocessors and with the 80960KB data sheet. Order Number: 270564-002.

- *80960SA/SB Reference Manual*

  This manual provides reference information applicable to the 80960SA/SB embedded processor. It is written for both software and hardware designers familiar with the principles of microprocessors and with the 80960SA/SB architecture. Order Number: 270929-002.

You might also find the following Intel data sheets useful:

- *80960CA-40, -33,-25, -16 32-Bit High Performance Embedded Processor*
  Order Number: 270727-003.
- *80960KB Embedded 32-Bit Processor with Integrated Floating-Point Unit*
  Order Number: 270565-004.
- *80960KA Embedded 32-Bit Processor*
  Order Number: 270775-002.
- *80960SA/80960SB Embedded 32-Bit Processors with 16-bit Burst Data Bus*
  Order Number: 270917-003.

To obtain Intel manuals and data sheets write to:

Intel Literature Sales
P.O. Box 7641
Mt. Prospect, IL 60056-7641

For phone orders in the U.S. and Canada call toll free: (800) 548-4725.

The following book is also a useful reference:

- Myers, Glenford J., David L. Budde, *80960 Microprocessor Architecture.* John Wiley & Sons. 1988.

  This book represents the most authoritative description of the 80960 processors and architecture. The authors explain why the designers choose a particular direction or added or omitted a particular function so that the user can understand and use the product better.

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W